COORDINATED STATIC AND DYNAMIC SCHEDULING FOR HIGH-QUALITY HIGH-LEVEL SYNTHESIS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Steve Haihang Dai

MAY 2019

© 2019 Steve Haihang Dai
ALL RIGHTS RESERVED

COORDINATED STATIC AND DYNAMIC SCHEDULING FOR HIGH-QUALITY HIGH-LEVEL SYNTHESIS

Steve Haihang Dai, Ph.D.
Cornell University 2019

The breakdown of Dennard scaling has led to the rapid growth of specialized hardware accelerators to meet ever more stringent performance and energy requirements. However, great performance-per-watt comes at the cost of enormous development effort. As the process of register-transfer level (RTL) optimization becomes unequivocally difficult, if not already unsustainable, high-level synthesis (HLS) has emerged as a promising approach to tackle the design productivity gap by raising the level of abstraction and in turn enabling lower design complexity and faster simulation speed. Scheduling forms the algorithmic core to state-of-the-art HLS technology, which automatically compiles untimed high-level (software) programs into cycle-accurate RTL (hardware) implementations. Given a set of constraints such as those arising from timing and resources, HLS scheduling extracts parallelism from the input program through control data flow analysis and determines the clock cycle at which each operation should be executed.

Despite increasing adoption of HLS for its design productivity advantage, the lack of success in achieving high quality-of-results (QoR) out-of-the-box continues to hinder the productivity advantage for which HLS is known. First, current scheduling algorithms rely heavily on fundamentally inexact heuristics that make ad-hoc local decisions due to scalability concerns with exact methods and cannot accurately and globally optimize over a rich set of constraints. This results in sub-optimal schedules for the generated hardware whose QoR gap remains unknown to the designer as well as the tool itself. Second, current scheduling techniques rely on static compiler analysis of the input program and must make simplifying assumptions about statically unanalyzable program behaviors. These assumptions are often too strong to provide adequate support for dynamic behaviors arising from variable-latency operations, irregular program patterns, and runtime hardware hazards. Third, HLS scheduling generates and uses inaccurate resource

and timing estimates that deviate significantly from actual post-implementation QoR. Inaccurate estimates prevent designers and the tool from performing meaningful design space exploration without resorting to the time-consuming downstream implementation process. The aforementioned challenges culminate in the algorithm, flexibility, and estimation gaps, respectively, faced by state-of-the-art HLS tools.

To tackle these major challenges of HLS, this thesis proposes a set of coordinated static and dynamic scheduling techniques to achieve QoR on-par with or exceeding that of manually optimized design. First, this thesis addresses the algorithm gap by improving static scheduling with a novel formulation based on system of integer difference constraints (SDC) and satisfiability (SAT) to exactly handle a variety of scheduling constraints. I develop specialized schedulers based on conflict-driven learning and problem-specific knowledge to optimally and efficiently solve scheduling problems leveraging modern constraint programming capabilities. Second, this thesis addresses the flexibility gap by proposing a set of dynamic scheduling techniques to synthesize flexible and complexity-effective HLS pipelines that are aware of dynamic structural and data hazards. I introduce scheduling and synthesis techniques to generate HLS pipelines with the ability to speculate, squash, and flush, making it possible to maintain high throughput in the presence of runtime hazards. Third, this thesis addresses the estimation gap by training a set of promising machine learning models to enable fast and accurate QoR estimation in HLS. The models are able to dramatically reduce estimation errors with negligible runtime cost.

BIOGRAPHICAL SKETCH

Steve Dai received his Bachelor of Science in Electrical Engineering from University of California, Los Angeles in 2011, and his Master of Science in Electrical Engineering from Stanford University in 2013. Shortly after, Steve joined the School of Electrical and Computer Engineering at Cornell University as a Ph.D. student in the Computer Systems Laboratory under the guidance of Professor Zhiru Zhang. Steve is broadly interested in cross-layer electronic design automation (EDA) and field-programmable gate array (FPGA), with an emphasis on enabling high-quality high-level synthesis (HLS) with novel optimization techniques and machine learning. Steve is also interested in investigating the intersection between HLS and hardware security, as well as developing HLS-based applications.

This dissertation is dedicated to my family, my teachers, and all who have helped and inspired me.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support, encouragement, advice, and help from many people here at and outside of CSL and Cornell. I would like to thank my family, teachers, mentors, friends, and all who have inspired me and helped me along the way to get to this point both academically, professionally, and personally.

First and foremost, I would like to thank my advisor and committee chair, Prof. Zhiru Zhang, for believing in me and inviting me to be one of the first students in the Zhang Research Group. This unique position allowed me to grow with the group and learn a lot more than just EDA. I would like to thank Zhiru for his unconditional support and constant availability throughout my time at Cornell and for giving me a fruitful Ph.D experience like no others.

I would also like to thank my minor committee member, Prof. Christopher Batten, for initially introducing me to Cornell and CSL, and most importantly to Zhiru. Without Chris, I would not have been aware of the array of opportunities available at CSL. In addition, I would also like to thank Chris for teaching me computer architecture in a systematic manner and providing great encouragement and valuable advice in our joint projects across architecture and HLS.

Furthermore, I would like to thank my other minor committee member, Prof. Edward Suh, for introducing me to the emerging field of hardware security and teaching me the key concepts in this area. I appreciate Ed's guidance in my initial efforts on trying to merge research in HLS and hardware security, and I am grateful to have the opportunity to contribute to the subsequent joint projects on security-constrained HLS between the Zhang Research Group and Suh Research Group.

This thesis was supported in part by NSF/Intel CAPA Award #1723715, NSF Awards #1149464, #1337240, #1453378, #1512937, #1618275, DARPA Award HR0011-16-C-0037, DARPA Young Faculty Award D15AP00096, ISRA Program under Intel Corp., Semiconductor Research Corp., and a research gift from Xilinx, Inc.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ALAP	as late as possible
ANN	artificial neural network
ASAP	as soon as possible
ASIC	application-specific integrated circuit
BRAM	block random-access memory
CDFG	control data flow graph
DAG	directed acyclic graph
DFG	data flow graph
DSL	domain-specific language
DSP	digital signal processing
EDA	electronic design automation
FF	flip-flop
FIR	finite impulse response
FPGA	field-programmable gate array
FSMD	finite state machine with datapath
HDL	hardware description language
HLS	high-level synthesis
II	initiation interval
ILP	integer linear programming
I/O	input/output
LUT	look-up table
MRT	modulo reservation table
PAR	place and route
QoR	quality of results
RRSE	relative root square error
RTL	register-transfer level
SAT	satisfiability
SCC	strongly connected components
SDC	system of integer difference constraints
SMT	satisfiability modulo theory
WNS	worst negative slack
XGBoost	Extreme Gradient Boosting

CHAPTER 1
INTRODUCTION

The breakdown of Dennard scaling has led to the rapid growth of specialized hardware accelerators to meet ever more stringent performance and energy requirements. However, great performance-per-watt comes at the cost of enormous development effort. While hardware specialization achieves significant improvement in performance, the process is way too slow to the point it is throttling progress [Sal16]. The traditional design flow is slow because of its reliance on the manual register-transfer-level (RTL) design methodology where designers must wrestle with low-level hardware description language and manually explore a large multidimensional design space. RTL requires a tremendous amount of effort and time to verify, and makes it difficult to re-target different design points because timing and microarchitecture are essentially fixed by design.

As the complexity of applications and hardware platforms continues to escalate, high-level synthesis (HLS) has emerging as an increasingly popular and promising alternative to traditional RTL design methodology to tackle the design productivity gap [CLN+11]. As the process of manual RTL optimizations becomes unequivocally difficult, if not already unsustainable, HLS provides the capability to automatically convert untimed high-level software programs into cycle-accurate RTL hardware implementations. By raising the level of abstraction from hardware to software, HLS enables lower design complexity and faster simulation speed. HLS reduces design effort while optimizing over a larger space of performance, area, and timing, and has been successfully applied in the design of accelerators [SDMZ17, AAHA+17a] to soft processors [SALL15]. High productivity allow HLS to enable shorter time-to-market, which is especially relevant in today's competitive and rapidly-evolving technology landscape. These benefits have led to growing development and adoption of both commercial and open-source HLS tools, including Xilinx Vivado HLS [CLN+11], LegUp HLS [CCA+11], Intel OpenCL [CNK+12], Intel HLS Compiler [Int18], Mentor Catapult HLS [Men19a], and Cadence Stratus HLS [Cad19b].

As shown in Figure 1.1, a typical HLS flow consists of three major stages: software compilation, scheduling and binding, and RTL generation. In the first stage, a high-level software program (e.g. C, C++, SystemC) is compiled and transformed into a control data flow graph (CDFG) using a software compiler such as GCC or LLVM. In the second stage,

scheduling analyzes and extracts parallelism from the CDFG to determine the clock cycle at which each operation should be executed given a set of timing and resource constraints that must be met. Binding interacts with scheduling to decide the instance of resource for which each operation should utilize. The scheduling and binding solution corresponds to a hardware model in the form of a finite state machine with datapath (FSMD). In the last stage, RTL model is generated based on the FSMD to implement the functionality described by the input software program. Within this process, scheduling provides the key algorithmic contribution to enable the transformation of an untimed sequential description with no concept of clock into a timed parallel implementation with pipeline registers. As such, scheduling is in a unique position to significantly influence the quality of the generated hardware.

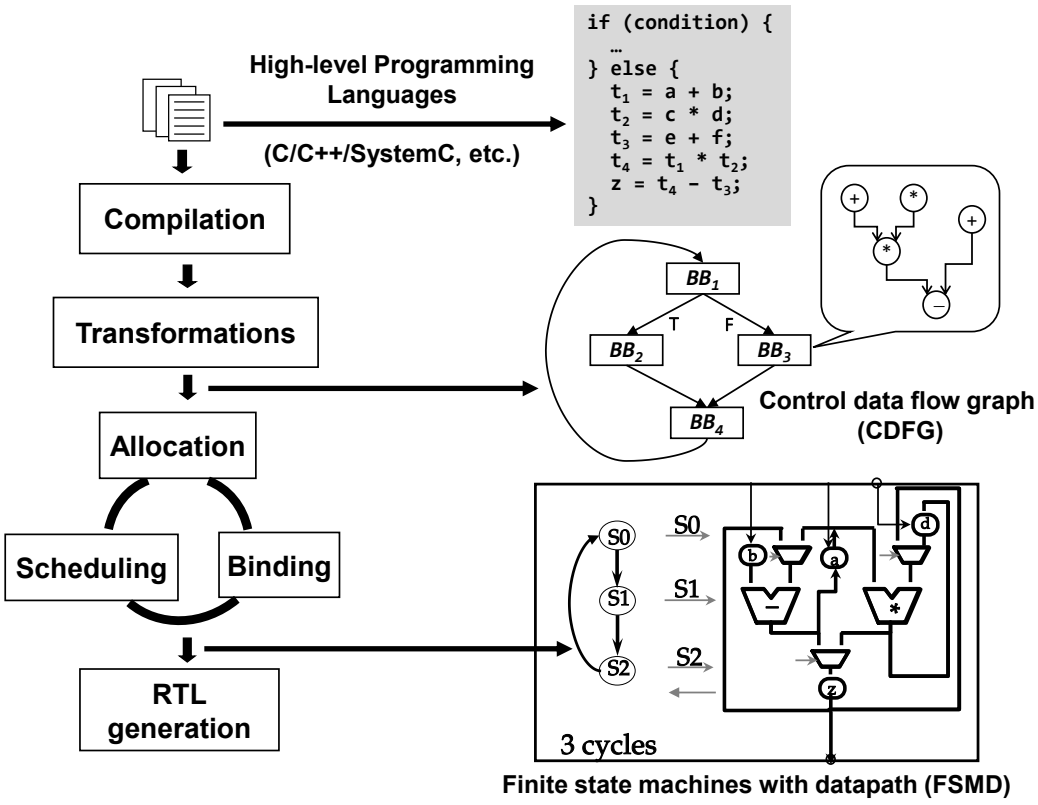


Figure 1.1: A typical HLS flow — A software program is synthesized into a hardware model through the three major stages of software compilation, scheduling and binding, and RTL generation.

HLS traditionally deals with the classic resource-constrained scheduling, an NP-hard problem which can be solved exactly with integer linear programming (ILP) but are

typically approximated using heuristics for better scalability. One such heuristic is list scheduling, a constructive algorithm that schedules ready operations one clock cycle at a time, based on an established priority and considering resource availability. It is a fast local optimization algorithm for minimizing latency under resource constraints, albeit suboptimally [De 94]. Compared to list scheduling, difference constraints based scheduling based on the system of integer difference constraints (SDC) is a more versatile heuristic for supporting operation chaining and additional design constraints such as relative input/output (I/O) timing [CZ06], and can be extended to pipeline scheduling [ZL13]. While it models resource constraints heuristically, SDC-based scheduling is rooted in a linear programming formulation and is more amenable to global optimization. Because of this advantage, SDC-based scheduling has been used in Vivado HLS [CLN+11] and LegUp HLS [CCA+11].

1.1 Current Challenges in HLS

Despite increasing adoption of HLS for its design productivity advantage, the lack of success in achieving high quality-of-results (QoR) out-of-the-box continues to hinder the productivity advantage for which HLS is known. According to a recent survey spanning 46 papers and 118 associated applications, HLS designs are clearly inferior in both performance and resource metrics. For example, only 39% of HLS designs attain performance better than or equivalent to that of corresponding RTL designs, and only 33% attain better or comparable usage of basic FPGA resources [LSVH18]. In this thesis, we attribute the existence of such a QoR gap to the *algorithm gap*, *flexibility gap*, and *estimation gap* inherent in today's HLS tools.

1.1.1 Algorithm Gap

A major challenge with current HLS lies in the fundamental inexactness of scheduling heuristics that are commonly used in HLS tools. For example, list scheduling makes ad-hoc local decisions. The popular SDC-based scheduling relies on heuristic ordering of resource-constrained operations to honor resource constraints, with no guarantee on

the quality of the heuristic ordering. As such, although scheduling heuristics are fast and scalable, they are fundamentally inexact with no guarantee on optimality. First, scheduling heuristics are designed to consider only a restrictive set of constraints and are unable to handle more complex scheduling problems. Second, they lack the ability to perform global optimization and may miss valuable optimization opportunities that can otherwise be discovered by exact techniques. In many cases, these challenges leave a quality-of-results (QoR) gap whose severity remains unknown to both the designer as well as the tool itself. This gap may be exacerbated as the quantity and variety of constraints increase for HLS to accommodate emerging application domains.

1.1.2 Flexibility Gap

Another major challenge stems from the fact that current HLS scheduling is forced to make assumptions, both optimistic and pessimistic ones. Because scheduling relies on static compiler analysis of the input program, it generates good schedules only for statically analyzable program patterns and must resort to making assumptions for compile-time unanalyzable, dynamic program behaviors. These assumptions are often too strong to provide adequate support for dynamic behaviors arising from variable-latency operations, irregular program patterns, and runtime data and control hazards. On one hand, HLS scheduling may optimistically assume that I/O delays are fixed while they can in fact vary from a single cycle to hundreds of cycles based on the memory hierarchy. On the other hand, HLS may pessimistically assume that load and store for the same array always alias in memory, when the alias pattern in fact depends heavily on the input data. These simplistic assumptions provide straightforward implementation for regular programs (e.g. digital signal processing) with well-structured execution patterns, but are neither accurate nor sustainable as HLS attempts to address irregular programs (e.g. graph algorithms, data analytics, sparse matrix computations) which cannot be fully analyzed statically because execution pattern is not known until runtime. These programs are characterized by data-dependent control flow, irregular memory dependence patterns, and dynamic workload. Without a way to accurately predict the runtime execution pattern of the program, static scheduling must make very conservative assumptions regarding

data and structural hazards and synthesize the least aggressive pipeline that trades performance for functional correctness.

1.1.3 Estimation Gap

There also exists a well-known disconnect between HLS and the downstream tool flow that has continued to frustrate both application designers and HLS tool developers alike. In particular, because HLS is unaware of the effects of a series of downstream transformations (e.g., logic synthesis, technology mapping, place and route (PAR)) on the design, QoR metrics generated and used in the HLS stage are highly inaccurate when compared to the actual QoR achieved post-implementation. Such miscorrelation on QoR between HLS and downstream stages prevents designers and the tool from applying the appropriate set of optimizations, resulting in designs with the wrong tradeoff. While it is always possible to iterate the downstream flow for every design point, doing so is impractical due to the notoriously slow runtime of implementation tools and would simply nullify the productivity advantage for which HLS is known in the first place. On the flip side, accurate estimation of post-implementation QoR at the HLS stage without running implementation is difficult because the final implemented design reflects the cumulative effects of many non-trivial transformations during implementation.

1.2 Thesis Overview

The numerous challenges of current scheduling approach leaves open a sizable QoR gap that is preventing the widespread adoption of HLS. It is evident that the inexactness of core scheduling algorithms, the lack of support for dynamic behaviors, coupled with the inaccuracy of QoR estimates form a multifaceted problem that must be addressed as it relates to *algorithm*, *flexibility*, and *estimation*. To tackle all these challenges, this thesis proposes coordinated static and dynamic scheduling, with contributions roughly divided into the following fronts:

New Static Scheduling to Improve Algorithm To eliminate the inexactness in HLS scheduling, this thesis improves static scheduling with a novel scheduling formulation based on SDC and satisfiability (SAT) to exactly handle a variety of scheduling constraints [DLZ18, DZ19]. I develop a specialized scheduler based on conflict-driven learning and problem-specific knowledge to optimally and efficiently solve scheduling problems. The solver takes advantage of the practical efficiency of modern SAT engines while exploiting the polynomial runtime of SDC algorithm to quickly prune the design space and scalably derive exact solutions. The practical scalability of the approach results in over 100x improvement in runtime over traditional ILP approach.

Dynamic Scheduling for Greater Flexibility To enable support for dynamic program and hardware behaviors, I develop a set of dynamic scheduling techniques to synthesize more flexible HLS pipelines that are aware of runtime hazards [DZL+17, DTHZ14]. This thesis introduces novel scheduling and synthesis techniques to generate pipeline with the ability to speculate, squash, replay, and flush. This thesis describes how I augment the default HLS synthesized pipeline with application-specific dynamic hardware logic to achieve high throughput in the presence of hard-to-predict runtime hazards while keeping the pipeline complexity-effective. For a range of representative irregular benchmarks, experiments demonstrate that dynamic hazard resolution for HLS can achieve over 60% latency reduction with reasonable area and minimal timing overhead.

Machine Learning to Enable Better Estimation To minimize the miscorrelation on QoR between HLS and downstream stages, this thesis also explores a machine-learning-based approach to fast and accurate estimation of QoR at the HLS stage without the need to run the subsequent implementation flow [DZZ+18]. This capability serves as a starting point for enabling rapid yet meaningful design space exploration for both the designer and the HLS tool. Specifically, I present a set of promising machine learning models that are able to efficiently and effectively bridge the accuracy gap for HLS QoR estimations. Experiments show that these models can reduce the errors of HLS estimations by up to 138% using features extracted exclusively from HLS.

This thesis' contributions center around the overarching goal of attaining the promise of high QoR for HLS by identifying critical gaps in HLS and addressing each one of them. Figure 1.2 illustrates the overall structure of this thesis and articulates the steps taken to reaching this promise — from breaking down the overall problem into specific challenges to developing solutions to address each challenge. First, this thesis improves exact HLS scheduling by pushing the limit of what is practically scalable, leveraging conflict-driven learning and problem-specific knowledge specially crafted for the HLS scheduling problem. Second, this thesis synthesizes application-specific pipelines that are high-performance, flexible, yet area-efficient by using intelligent static scheduling algorithms and augmenting the pipeline with dynamic hardware logic customized for the application. Third, this thesis leverages machine learning to improve EDA tools and demonstrates, in particular, its feasibility and benefit in improving QoR estimation accuracy for HLS. By pinpointing each gap, these contributions will deliver desirable improvements in out-of-the-box QoR for HLS tools.

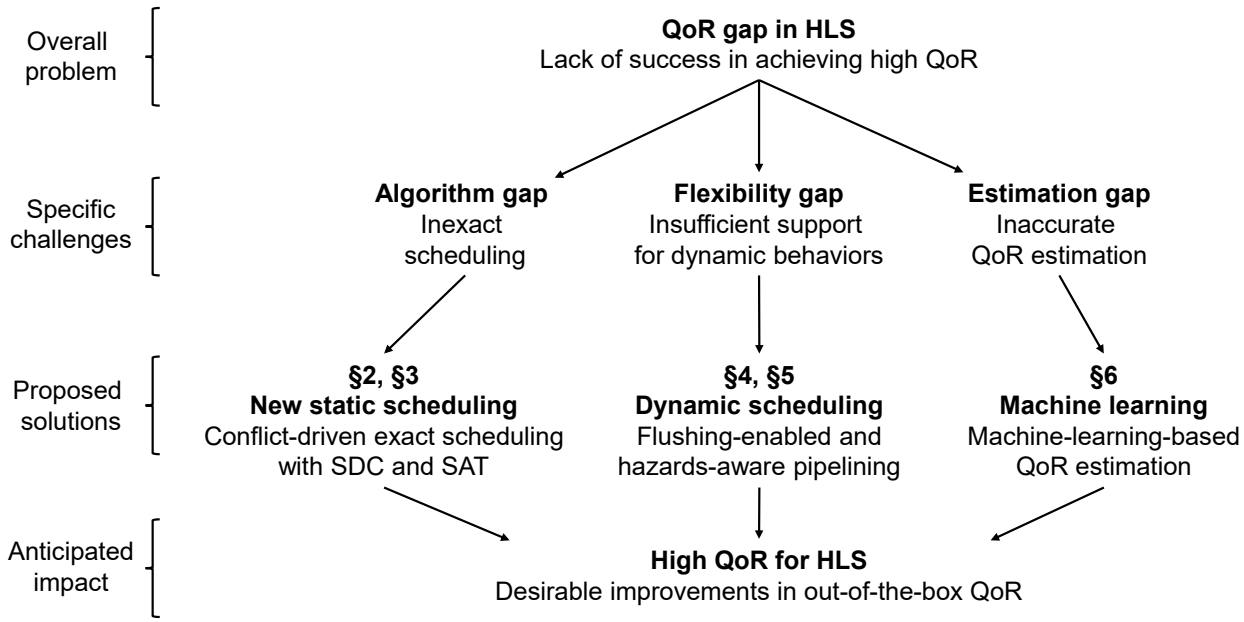


Figure 1.2: Overall structure of this thesis — This thesis breaks down the overall problem to specific challenges. Each challenge is addressed with the proposed solutions to attain the anticipated impact.

The rest of the thesis is organized as followed: Chapter 2 provides details on my new exact scheduling algorithm and framework with joint SDC and SAT. Chapter 3 extends

this framework to support pipeline scheduling. Chapter 4 describes techniques to enable structural and data hazard resolution to achieve dynamic HLS pipelining for irregular programs. Chapter 5 analytically and experimentally studies three promising approaches for supporting pipeline flushing in HLS. Chapter 6 illustrates my machine learning approach to enabling fast yet accurate QoR estimation for HLS. Chapter 7 summarizes this thesis with additional insights and future directions.

1.3 Collaboration, Previous Publications, and Funding

This thesis would not have been possible without the contributions from the colleagues across both the Zhang Research Group and Batten Research Group as part of the Computer Systems Laboratory under the School of Electrical and Computer Engineering at Cornell University, as well as outside industry collaborator from Xilinx, Inc. In particular, my advisor and committee chair Prof. Zhiru Zhang served as an integral part of all five projects presented in this thesis and provided detailed guidance from inception to completion for these projects as well as my general Ph.D research and career. In addition, my minor committee member Prof. Christopher Batten provided invaluable guidance for the Dynamic Hazard Resolution project (Chapter 4) from conceptualization to implementation, which contributed immensely to the realization of the project.

Gai Liu contributed to the initial research and the formulation for the SDS scheduler (Chapter 2). Ritchie Zhao, Shreesha Srinath, and Udit Gupta helped implement some of the benchmarks used in the Dynamic Hazard Resolution study (Chapter 4). Ritchie Zhao and Shreesha Srinath also created the PyMTL-HLS framework leveraged to perform experiments for this project and provided timely assistance in the usage of PyMTL. For the Flushing-Enabled Loop Pipelining project (Chapter 5), Dr. Mingxing Tan contributed to formulating various solutions to the problem, while Dr. Kecheng Hao helped implement and validate the proposed techniques on a commercial HLS tool. For the QoR estimation project (Chapter 6), I greatly appreciate the help of Yuan Zhou in prototyping the initial machine-learning-based training and testing framework, Qiang You for further improving this framework, and Hang Zhang for providing a pointer to XGBoost and exploring other machine learning models in an effort to close the QoR estimation gap.

This thesis was supported in part by NSF/Intel CAPA Award #1723715, NSF Awards #1149464, #1337240, #1453378, #1512937, #1618275, DARPA Award HR0011-16-C-0037, DARPA Young Faculty Award D15AP00096, ISRA Program under Intel Corp., Semiconductor Research Corp., and a research gift from Xilinx, Inc.

Before proceeding with the main contents of this thesis, I would like to briefly summarize other related publications that are not discussed in the rest of this thesis.

- The information flow constrained HLS project [JDSZ18] investigates the cross-section between HLS and information flow security and proposes a novel HLS framework to automatically generate hardware that guarantees the absence of both explicit and implicit information flow. I was involved throughout the process in formulating the security problem in the context of HLS and proposing techniques to close explicit and implicit information flow leveraging an HLS framework.
- The Celerity project [DXT+18, AAHA+17a, AAHA+17b] studies the architecture and design of an open-source 511-core RISC-V tiered accelerator fabric, where I participated in the design and synthesis of a binarized neural network accelerator targeting ASIC using the Stratus HLS tool.
- The Rosetta project [ZGD+18, GDZ15] and Face Detection project [SDMZ17] aim to develop a set of realistic HLS benchmark designs with actual performance constraints and advanced hardware optimizations to facilitate comparison between HLS tools, evaluate and stress-test new synthesis techniques, and establish meaningful performance baselines to track progress of the HLS technology. I contributed to initiating and building the groundwork for this project. In addition, I participated in the development of the face detection accelerator.
- The adaptive loop pipelining project [DLZZ17, LTD+17, TLZ+15, TLDZ14] serve as extensions to the dynamic pipelining techniques described in this thesis by developing novel architectures to enable high-throughput pipelining of irregular loops. I contributed to the formulation of various techniques and implemented different components of the project.
- The mapping-aware HLS scheduling project [ZTDZ15, TDGZ15] addresses the HLS timing estimation error due to the miscorrelation between HLS and the downstream

flow, which relates to the QoR estimation gap discussed in Chapter 6. Instead of leveraging machine learning, this project proposes to incorporate some degree of technology mapping in the HLS scheduling process to more aggressively pack operations into clock cycles. I was involved in formulating the algorithm to solve the mapping-aware scheduling problem.

CHAPTER 2
EXACT RESOURCE-CONSTRAINED SCHEDULING WITH JOINT SDC AND SAT

In the domain of static scheduling, there has always been an inherent tension between scalability and quality. Specifically for the resource-constrained scheduling problem, there has been a line of work that sacrifices either scalability or quality, as shown in Figure 2.1. On one hand, we have exact methods, such as ILP, that performs global optimization to achieve high-quality results at the cost of slow runtime. On the other hand, we have heuristics like list scheduling [PPM86] and SDC-based scheduling [CZ06] that make greedy decisions and sacrifice quality for scalability to various degrees. While SDC-based scheduling is implemented in state-of-the-art commercial and open-source HLS tools [CLN+11, CCA+11] and defines the current frontier between quality and scalability as evident in Figure 2.1, this section aims to push this boundary and develop a new scheduling formulation that achieves both optimal quality and fast runtime.

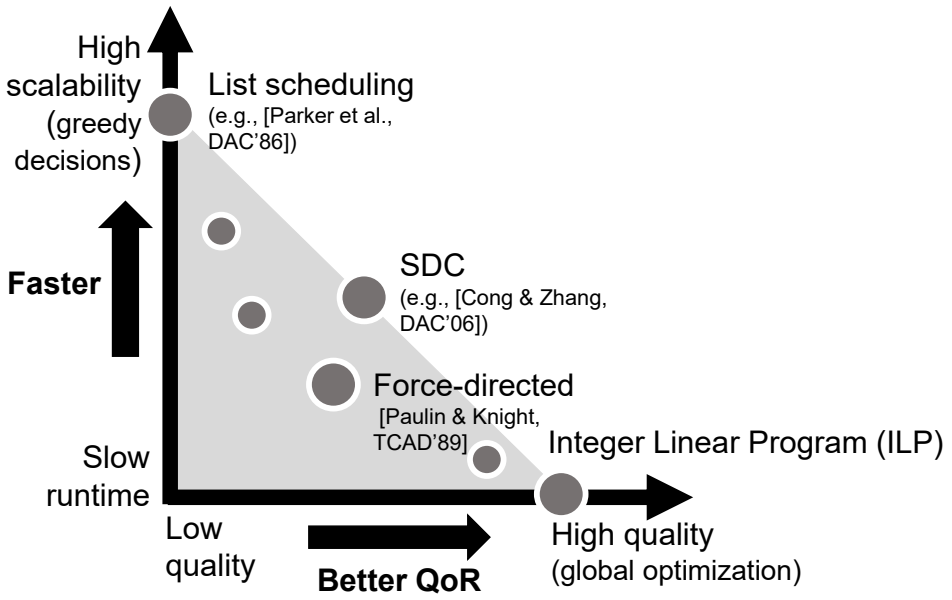


Figure 2.1: Scalability vs. quality tradeoff in scheduling.

In this chapter, I propose a scheduling formulation based on SDC coupled with SAT to exactly model a rich set of scheduling constraints. Inspired by satisfiability modulo theory (SMT) [DMB11], my proposed approach exploits the efficiency of SDC while leveraging

the scalability of modern SAT solvers to quickly prune away infeasible schedule space and derive optimal schedule. My scheduling technique aims to push the limit on what is practically scalable with exact scheduling as well as the variety of constraints that can be efficiently encoded and solved. The specific contributions of this chapter are as follows:

1. I propose a novel resource-constrained scheduling formulation, which combines SDC and SAT problems, to exactly and efficiently encode both resource and timing constraints in HLS.
2. I devise an exact yet fast resource-constrained scheduling algorithm for HLS based on conflict-driven learning by leveraging the efficiency of SDC and scalability of modern SAT solvers.
3. I employ problem-specific knowledge to specialize our scheduling algorithm to enable optimization and incremental scheduling techniques that further improve scalability.
4. I apply the specialized scheduler within the open-source HLS tool LegUp to efficiently synthesize high-quality RTL for a range of representative benchmarks targeting FPGAs.

The rest of this chapter is organized as follows: Section 2.1 provides background on scheduling and relevant theories, as well as motivation for the proposed approach; Section 2.2 details the new scheduling formulation; Section 2.3 describes the specialized conflict-driven scheduler; Section 2.4 presents experimental results; Section 2.5 provides related work and additional discussions.

2.1 Preliminaries

A typical HLS flow employs a software compiler (e.g., LLVM, GCC) to compile the input high-level program into a CDFG on which scheduling is then performed. In this section, we focus on the resource-constrained scheduling problem, which is also a classic optimization problem in operation research. In the context of HLS, the problem is described as follows:

Given: *(1)* A CDFG $G(V_G, E_G)$ where V_G represents the set of operations in the CDFG and E_G represents the set of edges; *(2)* A set of scheduling constraints, which may include dependence constraints, resource constraints, cycle time constraints, and relative timing constraints.

Objective: Construct a minimum-latency schedule so that every operation is assigned to at least one clock cycle while satisfying all scheduling constraints.

We illustrate the three types of scheduling formulation using the data flow graph (DFG) in Figure 2.2(a). As our running example, we would like to schedule the DFG targeting a clock period T_{clk} of `5ns`. We assume that each `add` or `store` operation incurs a delay of `1ns`, and each load operation incurs a delay of `3ns`. We further assume that only two memory read ports are available, so at most two `load` operations can be scheduled within the same cycle. `add` and `store` operations are unconstrained.

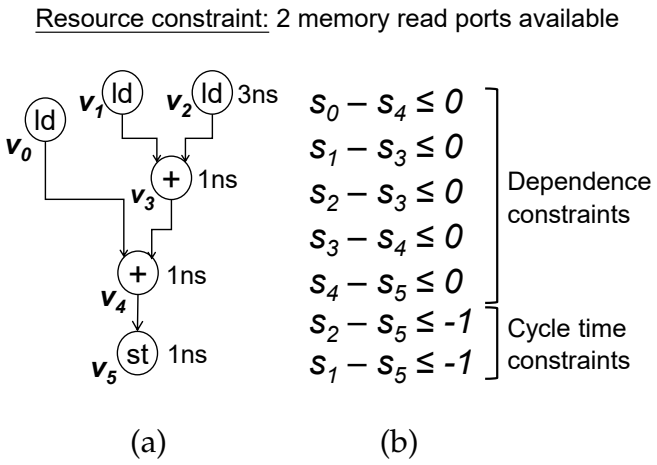


Figure 2.2: Motivational and running example for resource-constrained scheduling — *(a)* DFG for our example. Delay of each operation type is indicated next to the corresponding node. Resource constraint denotes that only two memory read ports are available. No resource constraints are imposed on `add` or `store` operations. *(b)* Dependence constraints and cycle time constraints corresponding to the DFG for a target clock period of `5ns`.

2.1.1 SDC-Based Formulation

SDC is a system of inequality constraints in the integer difference form $x_i - x_j \leq b_{ij}$, where b_{ij} is an integer, and x_i and x_j are variables. The system is feasible if there exists a

solution that satisfies all inequalities in the system. Because of the restrictive form of the constraints, SDC can be solved efficiently. For SDC-based scheduling [CZ06], a schedule variable s_i is declared for each operation i in the CDFG to denote the clock cycle at which operation i is scheduled. All SDC scheduling constraints are then expressed in the integer difference form so that the system consists of a totally unimodular constraint matrix over which an optimal integer solution can be guaranteed in polynomial time. For resource-constrained scheduling, we minimize the objective l such that $l > s_i \; \forall i$, where l represents the latency of the design.

To handle data dependence, SDC creates the following difference constraint for each data edge from operation i to operation j in G.

$$s_i - s_j \leq 0 \tag{2.1}$$

In our example, because there is an edge from node v_0 to node v_4, SDC will impose the difference constraint $s_0 - s_4 \leq 0$ to ensure that v_4 is scheduled no earlier than v_0. Similar constraints are constructed for other data dependence edges. To honor the target clock period T_{clk}, SDC identifies the maximum critical combination delay $D(ccp(v_i, v_j))$ between pairs of operations i and j and constructs the following different constraint to ensure that the combinational path with total delay exceeding the target cycle time T_{clk} must be partitioned into $\left\lceil D(ccp(v_i, v_j))/T_{clk} \right\rceil$ number of clock cycles.

$$s_i - s_j \leq -(\left\lceil D(ccp(v_i, v_j))/T_{clk} \right\rceil - 1) \tag{2.2}$$

In our example, because the maximum critical delay from v_2 to v_5 ($D(ccp(v_2, v_5)) = 6ns$) exceeds the target clock period of `5ns`, SDC will impose the constraint $s_2 - s_5 \leq -1$ to ensure that v_5 is scheduled at least one cycle after v_2. Similar constraints are imposed for combinational paths from v_1 to v_5 and v_0 to v_5. The aforementioned dependence and cycle time constraints are indicated in Figure 2.2(b).

While SDC is able to model timing constraints exactly, it must heuristically transform resource constraints into the integer difference form by imposing a particular heuristic linear ordering on the resource-constrained operations. This process separates resource-constrained operations appropriately into different cycles to ensure that sufficient resources are available to execute operations scheduled within the same cycle. The linear ordering consists of a set of precedence relationships between pairs of resource-constrained

operations i and j represented in the form of

$$s_i - s_j \leq -L_i \tag{2.3}$$

where L_i denotes the latency (in cycles) of operation i. Although the linear ordering results in a legal schedule that satisfies all resource constraints, the schedule is likely sub-optimal because the linear ordering is devised heuristically. There are many possible such legal linear orderings, some resulting in better schedules than others. However, SDC can simply pick one particular linear ordering heuristically and without knowledge of whether it is optimal.

For our example, SDC must impose partial orderings among the resource-constrained `load` operations because only two memory read ports are available for the three `load` operations (v_0, v_1, and v_2). On one hand, SDC can impose an edge from v_0 to v_1 as shown in bold in Figure 2.3(a) to separate v_0 and v_1 into different cycles so that each cycle has at most two `load` operations. With this heuristic partial ordering, the DFG requires at least three cycles to execute due to the critical path delay from v_0 to v_5. Given the target clock period of `5ns`, v_0 and v_1, each of which incurs a delay of `3ns`, must be scheduled in separate cycles given the partial ordering edge between them. v_5 cannot be scheduled in the same cycle as v_1 because there is no slack remaining in the clock cycle after scheduling v_3 and v_4. On the other hand, if SDC instead imposes an edge from v_1 to v_0 and another edge from v_2 to v_0 as shown in bold in Figure 2.3(c), the DFG can achieve a better latency of only two cycles while ensuring that each cycle has at most two `load` operations. In Figure 2.3, corresponding SDC constraints are shown in (b) and (d), respectively, with appended partial ordering ("resource") constraints boxed.

From this example, we see that it is necessary to enumerate all possible combinations of partial orderings and solve an SDC for each combination of imposed "resource" edges to find the optimal (minimum-latency) schedule. However, attempting all combinations is not scalable in the general case for an arbitrary number of resource-constrained operations. For this reason, SDC heuristically imposes one particular partial ordering without guarantee of optimality and proceed with solving the scheduling problem without regards to the effect of any sub-optimality on the solution.

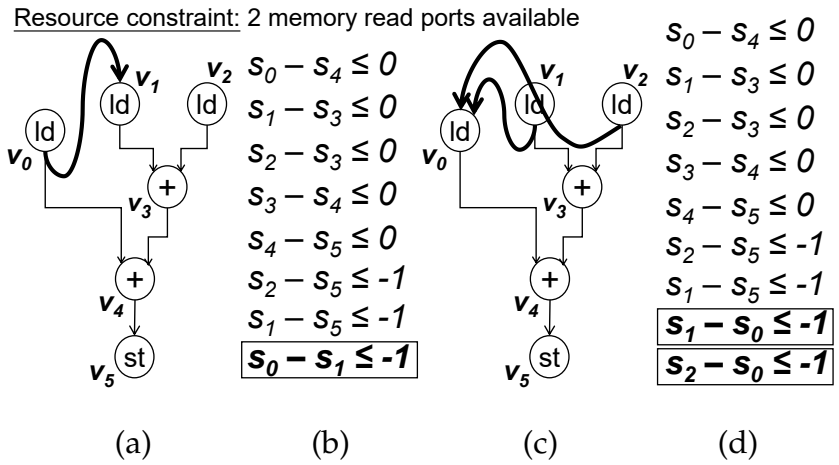


Figure 2.3: Partial ordering edges are heuristically imposed on the DFG, and subsequently in the SDC, to satisfy the resource constraints — Partial ordering edges are shown in bold, and corresponding difference constraints are boxed. *(a)-(b)* represent a different combination of partial ordering edges than *(c)-(d)*. Minimum latency differs depending on the particular combination.

2.1.2 ILP-Based Formulation

Applying ILP in the context of resource-constrained scheduling problem has been a well-studied topic [De 94]. ILP is a linear program with linear objective and constraints in which all variables are restricted to be integers. For the ILP-based formulation, we focus on the special case of 0-1 ILP in which all variables are binary. The formulation declares a binary variable x_{it} to denote whether operation i starts at clock cycle t, where i and t are integers bounded by the total number of operations and maximum allowable latency, respectively. With these binary variables, the start time s_i of operation i can be expressed as

$$s_i = \sum_{t=0}^{L-1} t \cdot x_{it} \tag{2.4}$$

where L denotes the maximum latency. Because s_i is analogous to the corresponding schedule variable in SDC, dependence constraints in ILP can be equivalently represented as the difference between pairs of schedule variables as in Eq. 2.1. For our example, we can safely assume a maximum start time equal to the number of operations $N = 6$. It follows that we declare variables $\{x_{00}, x_{01}, x_{02}, x_{03}, x_{04}, x_{05}\}$ for operation v_0 and denote that

$s_0 = \sum_{t=0}^{6-1} t \cdot x_{0t}$. Variables are similarly declared and derived for operations v_1 to v_5. The objective is same as that defined in Section 2.1.1 for the SDC formulation.

Unlike in SDC, resource constraints can be encoded exactly as linear constraints in ILP. To ensure that the number of active operations of type r in clock cycle t does not exceed the number of available type-r resources a_r, the ILP formulation imposes the resource constraint

$$\sum_{i:RT_i=r} \sum_{t'=t-L_i}^{t} x_{it} \leq a_r \tag{2.5}$$

where RT_i and L_i denote the resource type and latency of operation i, respectively. For our example, the ILP formulation needs to impose the constraints $\sum_{i=0}^{2} x_{it} \leq 2$ for each clock cycle t because only two memory ports are available. These constraints apply to the resource-constrained `load` operations v_0, v_1, and v_2 (i.e., $i = 0, 1, 2$). The second summation is omitted because the latency of `load` operation is zero-cycle in our example. The ILP formulation also requires the following unique start time constraint for each operation i to ensure that operation i starts at only one particular clock cycle.

$$\sum_{t} x_{it} = 1 \tag{2.6}$$

While modern ILP solvers can handle problems of non-trivial size with the branch and cu method [Mit02], ILP is in general NP-hard and difficult to scale. In comparison to SDC for scheduling, ILP requires significantly more variables for encoding the same problem and cannot take advantage of special matrix structure to efficiently solve the problem.

2.1.3 SAT-Based Formulation

SAT stands for the Boolean satisfiability problem, which determines if there exists an assignment of the Boolean variables that satisfies a Boolean formula. A SAT problem consists of a set of Boolean clauses, all of which must be satisfied by some assignment of the Boolean variables for the problem to be satisfiable. The problem is unsatisfiable otherwise. In general, a SAT-based scheduling formulation [Hor10] uses Boolean variable x_{it} to denote whether operation i starts at clock cycle t, and employs Boolean variable u_{it} to denote whether operation i is active at clock cycle t. Dependence and resource constraints can be expressed as clauses of these variables.

Modern SAT solvers perform systematic search based on variations of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [DLL62] of *decide*, *propagate*, and *backtrack*. These solvers recursively *decide* the value (true or false) of an unassigned variable, *propagate* the effects of this decision using deduction rules, and *backtrack* if conflicts dictate that a different value should be attempted for the variable. In particular, conflict-driven SAT solvers complements DPLL with extra features to achieve significant improvement in efficiency. Extra features may include clause learning, non-chronological backtracking, adaptive branching, unit propagation, and random restart [ZMMM01]. Although SAT remains a well-known NP-complete problem, SAT procedures based on the DPLL algorithm have demonstrated scalability with hundreds of thousands of variables and clauses [MZ09]. In the domain of design automation, SAT has been successfully applied to solve problems in hardware/software model checking, test pattern generation, equivalence checking, etc. However, it is interesting to note that although the scheduling problem can be encoded completely in SAT, the encoding is often too large and too inefficient even considering the capability of modern SAT solvers [Nie12]. Moreover, SAT is only concerned with whether the problem is satisfiable and does not inherently support optimization of an objective, such as minimizing latency.

2.2 Joint SDC and SAT Scheduling

As described in Section 2.1, the SDC heuristic achieves fast runtime but generates suboptimal schedule because resource constraints cannot be represented exactly with integer difference constraints. The ILP-based formulation can model both timing and resource constraints exactly but is not scalable in general. As a result, resolving the tension between scalability and quality is key to achieving both global optimization and fast runtime.

To this end, I propose a scheduling algorithm that integrates SDC and SAT to exactly handle different types of constraints and optimally solve the resource-constrained scheduling problem defined in Section 2.1. To achieve global optimization, my proposed algorithm leverages SDC to represent constraints that can be readily expressed in the integer difference form and employs SAT to encode constraints that do not naturally fall

under the SDC framework. A joint SDC and SAT formulation allows us to leverage the advantages of SDC and SAT while exactly encoding both timing and resource constraints.

Figure 2.4 shows the high-level structure of the proposed scheduler, mainly composed of a conflict-based SAT solver integrated with a graph-based SDC solver. On the left, the SAT solver takes advantage of conflict-based search (detailed in Section 2.2.1) to quickly propose partial orderings that satisfy the resource constraints. These partial orderings are converted to SDC constraints and appended to the SDC problem. On the right, the SDC solver leverages a graph-based algorithm (detailed in Section 2.2.2) to efficiently check the feasibility of the proposed partial orderings. Any infeasibility will be encoded as a conflict clause in SAT and appended back into the SAT problem. The solver iterates between SAT and SDC until it finds a feasible solution or proves that such solution does not exist.

Because a particular binding (set of partial orderings) proposed by SAT may not be consistent with the given SDC timing constraints, it is necessary to communicate any SAT binding decision to the SDC so that constraints in SDC and SAT are jointly considered. At the same time, any infeasibility must be communicated back from SDC to SAT so that SAT can learn from the mistakes of its previous proposals and make better proposals in the future. This process of conflict-driven learning is key to enabling accelerated convergence of the proposed scheduler. It is important to note that despite the benefits of conflict-driven learning, the problem remains NP-hard. Nevertheless, this approach demonstrates better efficiency and scalability than ILP. While the joint SDC and SAT approach is inspired by and bears resemblance to SMT, we will discuss the key differences in Section 2.5.

2.2.1 SAT for Resource Constraints

As shown in Figure 2.4, the proposed algorithm leverages SAT to model the resource constraints based on which partial orderings are proposed. In the SAT part of the formulation, let binding variable B_{ik} denote whether operation i is bound to resource instance k. We employ one binding variable to denote the binding of each resource-constrained operation to each resource instance. For our example, operations v_0, v_1, and v_2 are resource-constrained `load` operations, each of which can be bound to one of two memory read ports (i.e., $k = 0, 1$). Therefore, we declare $\{B_{00}, B_{01}, B_{10}, B_{11}, B_{20}, B_{21}\}$ for the different

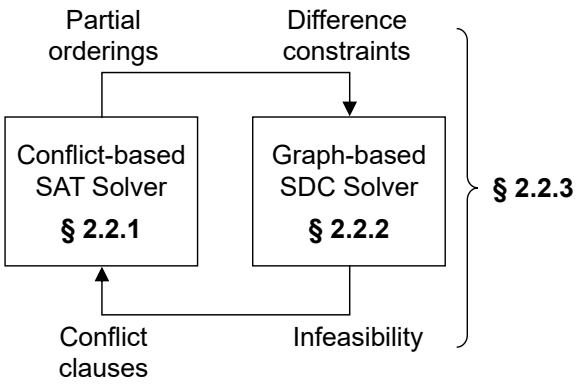


Figure 2.4: Overall structure of the proposed scheduler — Composed of a SAT solver integrated with an SDC solver to enable conflict-driven learning. This solver checks the feasibility of a particular latency. Latency optimization (Section 2.2.4) is built on top of this solver.

operation-resource pairs. By adding the appropriate clause $\sum_k B_{ik} = 1 \; \forall i$ to enforce that each operation is bound to exactly one resource, the binding variables are responsible for assigning each operation to a resource instance without exceeding the resource availability.

Based on the definition of binding variable, a sharing variable R_{ij} can be derived to denote whether operation i is sharing the same resource with operation j. For each pair of operations (i, j) mapped to the same type of resource,

$$R_{ij} = \bigvee_{k \in T} (B_{ik} \wedge B_{jk}) \tag{2.7}$$

where T denotes the set of resources of the particular type. R_{ij} is true if both operations i and j are bound to the same resource instance by the binding variable. With R_{ij}, we can then define the partial ordering variable $O_{i \to j}$ to denote whether operation i is scheduled in an earlier cycle than operation j. $O_{i \to j}$ maps to integer difference constraint in SDC between i and j as follows:

$$O_{i \to j} = True \;\mapsto\; s_i - s_j \leq -1 \tag{2.8}$$

$$O_{i \to j} = False \;\mapsto\; \emptyset \tag{2.9}$$

As shown in Eq. (2.8), assigning $O_{i \to j}$ to true dictates that operation i must be scheduled in an earlier cycle than operation j and therefore maps to the difference constraint $s_i - s_j \leq$

-1. As shown in Eq. (2.9), assigning $O_{i\to j}$ to false maps to an empty set of constraints, indicating that it is not necessary to impose any partial ordering between operations i and j because no particular partial ordering is required by the proposed resource binding. Given the mapping between SAT and SDC, the following partial ordering clauses are included in SAT for each pair of operations (i, j) mapped to the same type of resource.

$$R_{ij} \to (O_{i\to j} \vee O_{j\to i}) \tag{2.10}$$

$$\neg(O_{i\to j} \wedge O_{j\to i}) \tag{2.11}$$

Eq. (2.10) indicates that if operation i and j shares the same resource instance, it implies that operation i must be scheduled either in an earlier cycle or in a later cycle than operation j. Eq. (2.11) ensures that operation i cannot be simultaneously scheduled both in an earlier cycle and later cycle than operation j.

Figure 2.5(a) shows the partial ordering clauses for our example problem where a pair of clauses is specified for every combination of resource-constrained `load` operations (v_0, v_1, and v_2). Among other types of clauses described, only the partial ordering clauses are shown because they contain the partial ordering variables to be mapped to SDC. In this figure, for example, the first clause indicates that if v_0 and v_1 share the same resource instance, v_0 must be scheduled either in an earlier cycle or in a later cycle than v_1, and not both. A similar line of logic follows with the other clauses in the figure. SAT clauses like these (e.g., Eq. (2.7), (2.10), (2.11)) can be translated into conjunctive normal form commonly accepted by SAT solvers. Subsequently, the resulting assignments of $O_{i\to j}$ and $O_{j\to i}$ satisfying these clauses will be mapped to integer difference constraints or lack thereof in SDC based on Eq. (2.8) and (2.9). For instance, $O_{0\to 1}$ assigned to true will be mapped to $s_0 - s_1 \leq -1$.

2.2.2 SDC for Timing Constraints

As shown in Figure 2.4, the proposed algorithm uses SDC to solve the difference constraints, which consist of incoming partial ordering constraints from SAT and the original set of timing constraints (e.g., dependence and cycle time constraints) of the problem previously shown in Figure 2.2(b) and reproduced for convenience in Figure 2.5(b). From Figure 2.5(b), we see the difference constraints can be conveniently represented using a

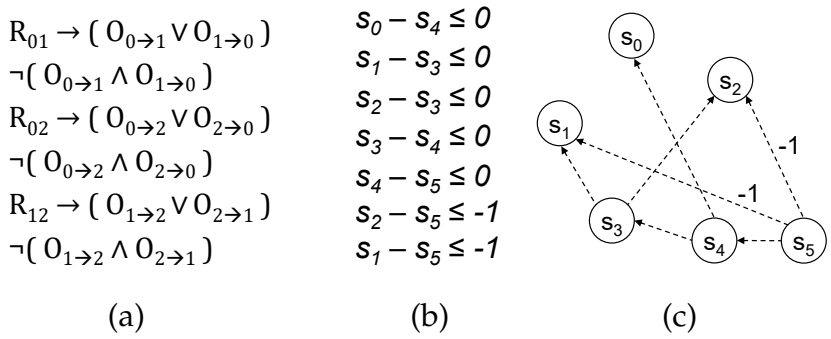


Figure 2.5: Constraints for our running example — *(a)* Resource constraints in SAT. *(b)* Timing constraints in SDC. *(c)* Corresponding SDC constraint graph.

constraint graph where each variable maps to a node and each constraint maps to an edge. The constraint graph contains edges to represent dependence constraints and cycle time constraints. Inequalities whose right-hand side is `0` represent dependence constraints, while those whose right-hand side is `-1` represent cycle time constraints, both described in Section 2.1.1. For each of these constraints in integer difference form $s_u - s_v \leq d_{u,v}$, the constraint graph includes an edge of weight $d_{u,v}$ from node v to u. For clarity, weights are omitted for zero-weight edges.

By representing SDC as a constraint graph, we can detect infeasibility of the difference constraints by the presence of negative cycle in the graph. This property will be useful for checking whether the proposed partial orderings from SAT are consistent with the given SDC timing constraints. In addition, the negative cycle serves as a certificate of any inconsistency between the proposed resource binding and given timing constraints. Section 2.2.3 will describe how to leverage the negative cycle to provide feedback from SDC to SAT for enabling conflict-driven learning. Furthermore, we can obtain a feasible schedule, either as late as possible (ALAP) or as soon as possible (ASAP) schedule, by solving a single source shortest path problem on the graph. ASAP schedules all operations to the earliest possible clock cycle, and ALAP schedules all operations to the latest possible clock cycle given a latency constraint.

In the proposed solver, it is necessary to detect whether the addition of each partial ordering edge induces a negative cycle in the constraint graph. However, it is wasteful to solve the entire SDC with all nodes and edges for each edge added when only a small

part of graph is affected by the addition. Doing so cuts directly into the bottom line of our scheduler because SDC is a crucial component of conflict-driven learning. Quick propagation and convergence of the scheduler rely on having a highly efficient SDC solver and a method to quickly identify any negative cycle in the constraint graph. To accelerate the process of conflict identification in SDC, I propose to leverage an efficient incremental algorithm for maintaining a feasible solution and detecting negative cycle for a dynamically changing SDC constraint graph [RSJM99].

To enable incremental SDC solving, the proposed scheduler initializes with a feasible solution (shortest path solution) of the original graph (without partial ordering edges). For each edge added to the constraint graph or each tightened edge weight, the algorithm traverses only the affected subgraph and update the distances of only affected nodes. This incremental update guarantees that the updated node values continue to maintain a feasible solution. Because the algorithm is essentially applying Dijkstra's algorithm [DV15] to modify only affected edges and nodes, the addition (or tightening) of a constraint incurs a marginal time complexity $O(\Delta e + \Delta v \log \Delta v)$, where Δe and Δv denote the number of affected edges and nodes, respectively. The algorithm is able to delete or relax an edge in constant time. Because deletion or relaxation results in a less constrained system, the current feasible solution remains feasible.

Using the incremental SDC algorithm, the scheduler inserts one edge at a time until the constraint graph becomes infeasible. The algorithm detects such infeasibility when the distance of the source node of the inserted edge is updated during the traversal of the affected subgraph. This indicates a negative cycle in the affected subgraph because the distances of the nodes will continue decrease as long as we continue to traverse the subgraph. At this point, the proposed algorithm traces backward on the predecessors along the shortest path computed by Dijkstra's algorithm to extract the edges involved in the negative cycle. The algorithm then reports partial ordering edges in the negative cycle back to SAT because SAT is concerned with resource-related partial orderings. Other edges represent hard constraints and are not influenced by SAT.

2.2.3 Conflict-Driven Learning

As shown in Figure 2.4, SAT and SDC interact closely within a feedback loop to enable conflict-driven learning. For each iteration of the loop, SAT proposes partial orderings that satisfy the SAT clauses described by Eq. (2.10) and (2.11). These partial orderings are converted to SDC constraints based on Eq. (2.8) and (2.9) and appended to the SDC problem. SDC then checks the feasibility of the proposed partial orderings and report any infeasibility as a conflict clause back to the SAT.

Figure 2.6 illustrates the power of conflict-driven learning using our running example. Here we would like to determine if the DFG in Figure 2.2(a) can be scheduled within two cycles. The corresponding SAT formulation for resource constraints is reproduced on the top of Figure 2.6(a), while the initial SDC constraint graph for timing constraints is shown on the bottom. As the solver progresses, resource-related edges mapped from the partial ordering variables will be added to the constraint graph in a manner similar to that of timing constraints described in Section 2.2.2. It is important to note that the constraint graph contains a latency edge of weight `1` from s_0 to s_5 to indicate a maximum allowable clock cycle index of `1` for our target two-cycle schedule starting with cycle `0`.

To solve the feasibility problem of determining whether the graph can be scheduled within two cycles, SAT starts with an initial proposal of the assignment of the partial ordering variables as shown on the top of Figure 2.6(b). For clarity, we show only partial ordering variables that are assigned to `True` because they are the ones that will influence the constraint graph. On the bottom of the figure, SDC adds the corresponding edges (shown with solid lines) proposed by SAT into the constraint graph. With these additional edges, SDC detects a negative cycle (shown in bold) among the initial edges and the partial ordering edge from $O_{0\to1}$. SDC then reports the conflict back to SAT using the conflict clause $\neg O_{0\to1}$ to ensure that any partial ordering involving v_0 before v_1 should no longer be proposed by SAT. As shown in Figure 2.6(c), after the conflict clause is added to the SAT problem, SAT makes a different proposal based on the updated set of clauses. In this case, SDC detects a different negative cycle involving the edge proposed by $O_{0\to2}$ and adds the conflict clause $\neg O_{0\to2}$ to the SAT. During conflict-driven scheduling, a negative cycle indicates that the resource binding proposed by SAT is inconsistent with the (hard) timing constraints of the problem. No schedule is able to achieve the desired latency

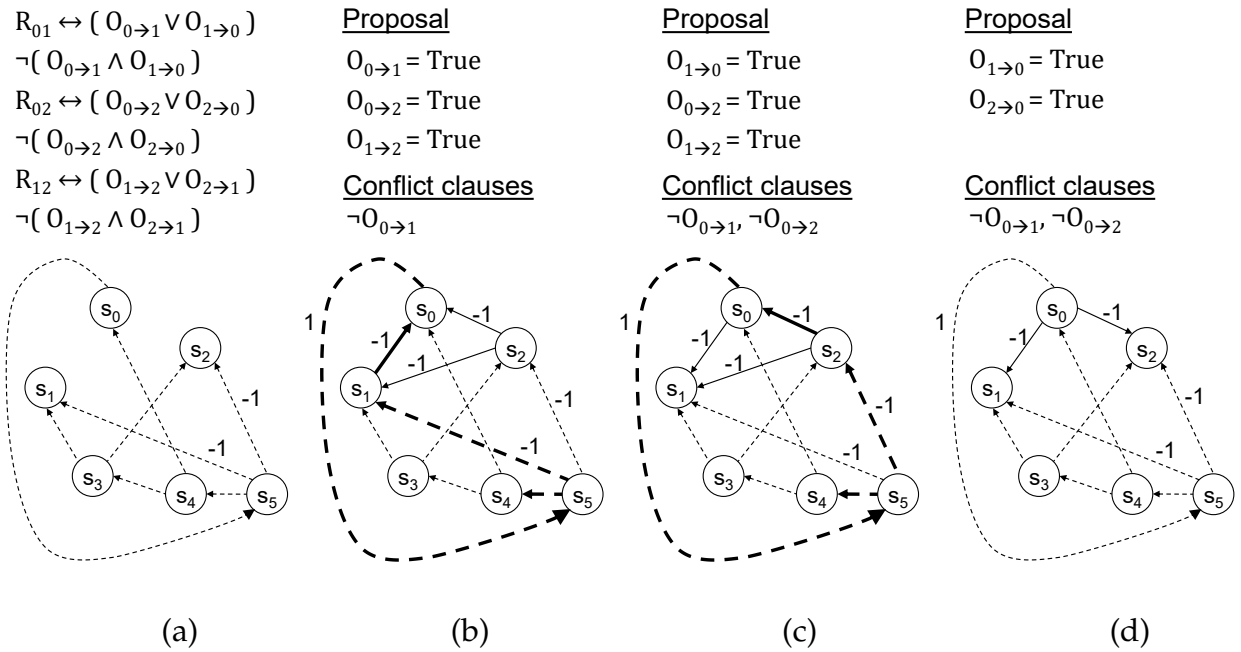


Figure 2.6: Illustration of conflict-driven learning with SDC and SAT using the running example from Figure 2.2 — *(a)* Resource constraints in SAT on the top and initial SDC constraint graph on the bottom. *(b)-(d)* The progression of joint SAT and SDC scheduling. Corresponding partial ordering proposals by SAT are shown on the top. For conflict clauses, ¬ denotes negation of the SAT variable. For constraint graphs, dashed lines represent hard constraints. Solid lines represent partial ordering constraints proposed by SAT. Bold lines trace negative cycles.

while satisfying both the timing constraints and the proposed resource binding. As a result, a different resource binding needs to be attempted.

Based on the feedback up until this point from SDC, conflict clauses dictate that any schedule with v_0 before v_1 or v_0 before v_2 will be infeasible and need not be attempted. Notice that these conflict clauses are short, allowing SAT to prune out a large search space because it no longer needs to propose any combination involving these infeasible orderings. Shorter conflict clauses lead to a larger search space that can be pruned and therefore faster propagation and convergence for our scheduler. As such, it is crucial to derive conflict clauses that are as short as possible. Negative cycle satisfies this property because it is guaranteed to be an irreducibly inconsistent set of constraints [VL81]. It is a minimal

set of inconsistent constraints in which the removal of any edge in the negative cycle will also remove the negative cycle in its entirety.

With two short conflict clauses, SAT has a much better understanding of the search space. As shown in Figure 2.6(d), SAT now makes a proposal whose corresponding edges no longer generate any negative cycle in the constraint graph. Because the constraint graph is now feasible, SDC returns a feasible solution that satisfies all timing and resource constraints. For efficiency, the scheduler uses the shortest path distances of the constraint graph as the feasible solution because the shortest path has already been computed in the process of detecting negative cycle.

2.2.4 Minimizing Latency

Because SAT has its root in decision problems, we have so far limited our discussion to checking the feasibility of a particular latency value. To minimize latency as in the case of resource-constrained scheduling, I propose to perform binary search over the range of possible latency values based on an initial upper and lower bound. During the binary search, the algorithm solve a series of feasibility problems as described in Section 2.2.3, each of which returns either a feasible solution or a proof that the problem is infeasible. A feasible answer allows the scheduler to decrease the upper bound, while an infeasible answer requires increasing the lower bound. The binary search terminates when the upper and lower bounds coincide.

Because the convergence of the scheduler depends on the number of latency values the binary search needs to process, I propose to leverage specialized knowledge that we can obtain for the scheduling problem to establish upper and lower latency bounds to reduce the range of latency values that need to be searched. Specifically, I propose to leverage the original SDC heuristic scheduling algorithm [CZ06] for upper bounding to establish a good initial solution that has already globally optimized over a subset of constraints. Furthermore, we propose to apply the resource-aware lower bounding algorithm [RJ94] (described later in Section 2.3.1) to establish a lower bound so that the scheduler does not waste time exploring too many unmeaningful latency values. While the upper and lower bounds are not necessarily tight, they provide a good starting point from which exact scheduling can initialize.

2.3 Scheduler Specialization

As mentioned in Section 2.2.4, it is possible to extract knowledge we have specific to the resource-constrained scheduling problem to further reduce the search space and improve runtime. In this section, we describe how to leverage various heuristics to specialize our scheduler for the scheduling problem. These techniques maintain the exactness of the algorithm and the optimality of the solution.

2.3.1 Resource-Aware Lower Bounding

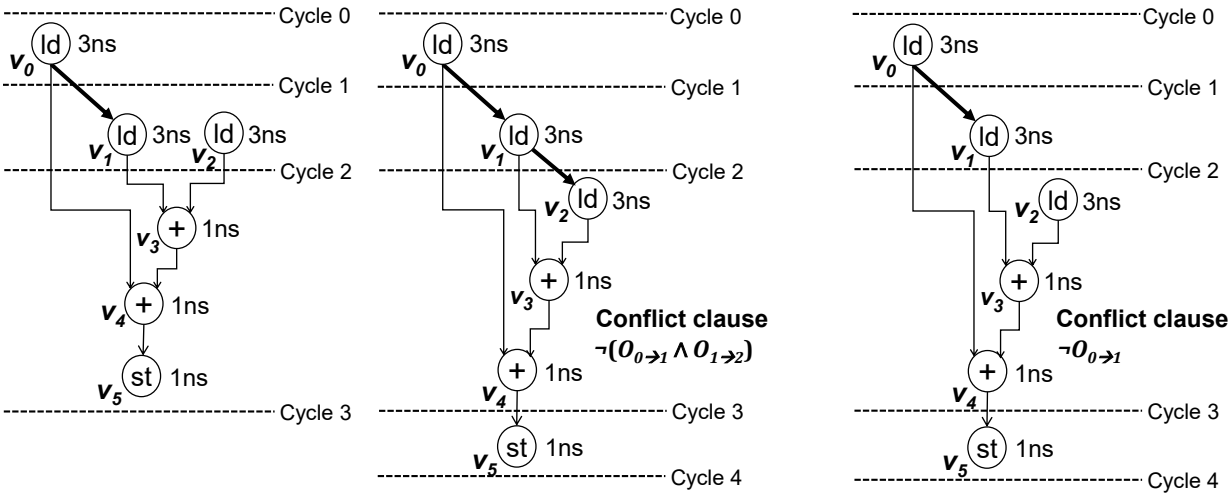
Resource-aware lower bounding applies a greedy algorithm to solve a relaxed version of the resource-constrained scheduling problem [RJ94]. While the algorithm eliminates dependence constraints for the relaxation, it uses the ASAP schedule to determine the earliest clock cycle each operation can be scheduled and minimizes the tardiness of each operation in respect to the ALAP schedule. The greedy algorithm selects the operation with minimum ALAP value and assigns it to the earliest clock cycle based on the ASAP schedule and resource constraints. This process continues until all operations have been scheduled. The resulting lower bound is determined by adding the maximum tardiness (in cycles) among all operations to the critical path latency for the entire design, which considers only dependence.

While we have discussed in Section 2.2.4 the application of resource-aware lower bounding to establish tighter lower bound in optimization, the same exact algorithm can be helpful for accelerating the propagation for conflict-driven learning described in Section 2.2.3. Recall that partial ordering edges are inserted one-by-one into the SDC constraint graph until the graph becomes infeasible. The fewer the number of inserted partial ordering edges, the shorter the conflict clause and larger the search space that can be pruned by SAT based on the conflict clauses. In addition to detecting negative cycle, the proposed scheduler can also incrementally determine the lower bound upon the insertion of each new edge. After identifying the first edge that results in an infeasible system, our scheduler uses the deletion filtering algorithm [CD91] to remove previously added edges that do not contribute to the infeasibility. An edge does not contribute to the infeasibility if the graph remains infeasible even after the edge has been removed.

The remaining set of edges then compose an irreducibly inconsistent set of constraints. Because the lower bounding algorithm is aware of the limited resource availability, it is actually able to prove infeasibility, in certain cases, with fewer partial ordering edges than SDC which has no sense of resource constraints other than those imposed by partial ordering. As such, lower bounding improves solution space pruning during conflict-driven learning.

We illustrate one such case in Figure 2.7 with the same DFG as in Figure 2.2(a). Here we would instead like to determine if the DFG can be executed within three cycles, assuming one memory read port and a target clock period of `5ns`. To separate the resource-constrained `load` operations (v_0, v_1, and v_2) into different cycles due to the availability of only one read port, let's further assume that partial ordering edges are added in the order corresponding to partial ordering variables $\{O_{0\to1}, O_{1\to2}\}$. In Figure 2.7, note that edges are shown within the DFG instead of the constraint graph. With the first partial ordering edge from v_0 to v_1 in Figure 2.7(a), SDC is unable to rule out the feasibility of executing the DFG in three cycles. Because SDC is unaware of the number of available read ports, it schedules v_2 in the same cycle as v_1. Only with the second edge from v_1 to v_2, as shown in Figure 2.7(a), does SDC pushes v_2 to the next cycle and realize that the DFG requires at least four cycles. Because the DFG cannot complete in three cycles with the two edges, SDC will return the conflict clause $\neg(O_{0\to1} \wedge O_{1\to2})$ to reflect the irreducibly inconsistent set of two edges. SDC requires both partial ordering edges (the complete resource binding) to decide infeasibility.

With resource-aware lower bounding, however, it is possible to determine that the DFG requires at least four cycles after adding only the first partial ordering edge from v_0 to v_1. As demonstrated in Figure 2.7(b), the algorithm does not attempt to schedule v_2 in the same cycle as v_1 even without the second edge, because the algorithm is aware that only one read port can be used in each cycle. As a result, v_2 is pushed to the next cycle, increasing the latency to at least four cycles. With lower bounding, the scheduler generates a more concise conflict clause $\neg O_{0\to1}$ for this example, which enables more effective pruning of the search space in SAT. Lower bounding is able to determine infeasibility with only a partial resource binding, thus resulting in speedup.



(a) SDC (b) Lower bounding

Figure 2.7: Illustration of the advantage of lower bounding over SDC in conflict-driven learning — Assume one memory read port and $T_{clk} = 5ns$. Actual DFGs, instead of constraint graphs, are shown in these figures. *(a)* SDC requires two "resource" edges (in bold) to determine that the DFG requires at least 4 cycles. *(b)* The lower bounding algorithm requires only one edge to determine the same 4-cycle latency because it pushes v_2 to the next cycle due to resource constraint.

2.3.2 Incremental Learning

Because the proposed SAT formulation in Section 2.1.3 includes variables for all resource-constrained operations, conflict-driven learning described in Section 2.2.3 considers all resource-constrained operations equally. In reality, however, some operations tend to be located in congested region of the schedule and must compete for a very limited number of resources within a limited number of time steps. Other operations do not fall in the congested region and can be freely scheduled. The congested region constitutes the problematic part of the schedule because there are more operations that need to be scheduled than the number of available resources for these operations. As a result, it would be more effective to emphasize the SAT's resource constraints over operations that are likely to encounter resource contention and allow non-contending operations to be scheduled by SDC's (hard) timing constraints only. This approach attempts to reduce the

size of the NP-hard part of the problem and leverages SDC as much as possible in finding a feasible schedule.

To implement this idea, I propose an incremental learning mode for the scheduler. Incremental learning leverages problem-specific knowledge to specifically target operations that are likely to cause resource contention. The flow of incremental learning mode is shown in Figure 2.8. Based on this flow, the scheduler starts with an empty SAT formulation, with no resource constraints initially. The scheduler then performs conflict-driven learning by propagating SDC and/or lower bounding (denoted as LB in the figure) with SAT. If the SDC graph reports a negative cycle, the problem is not satisfiable even with only timing constraints. In this case, the solver returns unsatisfiable and terminates. If the SDC graph does not detect any negative cycle, which is the more likely scenario, the scheduler checks the legality of the schedule against resource constraints. If the schedule is legal, the scheduler returns with the feasible schedule. If the schedule is illegal, likely in the initial iterations of this flow because no resource constraints have been considered, the scheduler will extract the contending operations with a list scheduling like heuristic. During the extraction process, the heuristic attempts to reorder operations to remove resource contention. If the heuristic succeeds in removing all resource contention, the scheduler also returns a feasible schedule. If contention remains, however, the scheduler adds the clauses of those contending operations to the SAT and repeats the flow starting with another iteration of conflict-driven learning.

The ultimate goal of incremental learning is to dramatically reduce the search space and improve runtime by using well-known heuristics (e.g., list scheduling, SDC-based scheduling) to direct the search toward the more difficult region of the schedule. Nevertheless, it is important to emphasize that these heuristics are used simply to guide the solver in a more promising path toward the solution and should in no way jeopardize the exactness of the scheduler. When incremental learning returns satisfiable, it always provides a legal schedule in regards to both timing and resource constraints and satisfies the given latency bound. Incremental learning is performed for different latency bounds in the binary search manner described in Section 2.2.4 to determine the schedule with the optimal latency.

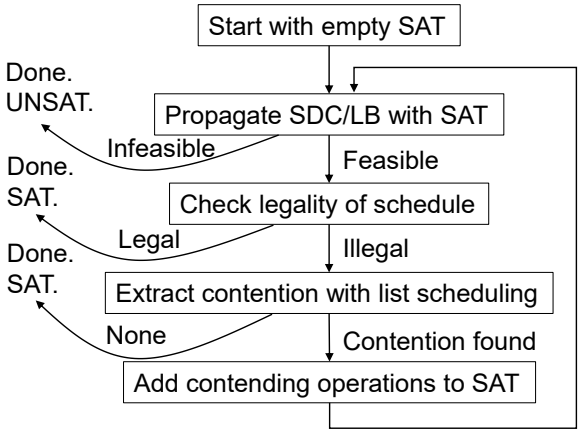


Figure 2.8: Incremental learning flow — Starts with no resource constraints and incrementally imposes resource constraints on operations that have encountered resource contention in previous iterations of the loop in this flow.

2.4 Experiments

The proposed scheduler (detailed in Section 2.3) is implemented in C++ interfaced with LLVM compiler and Lingeling SAT solver [Bie13]. We execute our scheduler on an Intel Xeon CPU running at 2.50GHz, and evaluate it on a set of compute-intensive benchmarks listed in Table 2.1. These benchmarks include a chemical plant controller and a number of DSP algorithms such as discrete cosine transforms. We constrain the scheduling process such that these benchmarks contain a large portion of resource-constrained operations useful for stress-testing our scheduler.

The first set of experiments aim to compare the runtimes of the proposed scheduler against those of state-of-the-art commercial and open-source ILP solvers. A comparison of runtime results between the proposed joint SDC and SAT scheduling (SDS for short) and default ILP scheduling is shown in Table 2.1. For the SDS scheduler, we provide results for the scheduler running in non-incremental mode and in incremental mode. `Non-incremental` column provides results from applying conflict-driven learning from Section 2.3.2 with the full set of SAT variables. `Incremental` column provides results from applying incremental learning from Section 2.3.2 by selectively targeting a subset of SAT variables. For default ILP scheduling, the formulation presented in Section 2.1.2 is solved in CPLEX [IBM17], a state-of-the-art commercial ILP solver, as well

as in CBC [For05], a best-in-class open-source ILP solver. Speedup values achieved by non-incremental and incremental modes against each ILP solver are shown respectively in parentheses in the corresponding columns.

Based on the results in Table 2.1, SDS scheduler running in non-incremental mode is faster than the open-source ILP solver by around two orders of magnitude and sometimes three orders of magnitude in all cases except `CHEM` for which both solvers time out. In non-incremental mode, SDS scheduler can also beat the commercial ILP solver by at least one order of magnitude, and up to two or three orders of magnitude for the same set of benchmarks. These results demonstrate the effectiveness of setting upper and lower latency bounds and exploiting negative cycle and lower bounding in propagation to quickly prune out the entire search space. It is interesting to note that benchmark `U5ML` achieves a low runtime because it is much more constrained by timing than by resource. Timing constraints dictate that its latency cannot be further reduced regardless of resource assignment.

With incremental mode enabled, Table 2.1 shows that SDS scheduler is able to complete the previously difficult benchmark `CHEM` and locate the optimal solution while both the commercial and open-source solvers struggle and time out. At the same time, incremental mode also improves the runtime of other benchmarks by various degrees. The improvement from incremental mode stems from the fact that only a small fraction of operations are actually involved in resource contention. Based on Table 2.1, mostly less than 10% of the SAT variables are needed to resolve resource constraints and converge to an optimal solution. The percentage becomes small for large benchmarks. With problem-specific knowledge, we specifically target contending operations to achieve significant speedup.

Table 2.2 shows the runtime in seconds for different combinations of constraints on the number of multipliers and memory ports. In general, an increase in the number of resources leads to additional SAT binding variables while the number of SAT partial ordering variables remains unchanged. The overall increase in the number of SAT variables may lead to longer runtime for the SAT solver. However, increasing the number of resources also loosens the resource constraints and decreases the number of partial ordering edges that needs to be inserted into the SDC (when the solver is running in in-

Table 2.1: Runtimes are reported in seconds for the proposed joint SDC and SAT scheduling (SDS for short) compared to default ILP scheduling using CPLEX and CBC — `%variables`: percentage of variables in non-incremental mode activated in incremental mode; speedup of `Non-incremental` and `Incremental` shown respectively in parentheses against `CPLEX` and `CBC`. `TO`: timeout after 300 seconds. `n/a`: not applicable. `Optimal Latency`: optimal latency in clock cycles for each benchmark and represents the latency achieved by both SDS scheduler and default ILP scheduling. `LegUp Latency`: latency achieved by LegUp using SDC-based scheduling heuristic.

Benchmark	# Operations	Runtime for SDS Scheduling (sec)		Runtime for Default ILP Scheduling (sec)		Optimal Latency	LegUp Latency
		Non-incremental	Incremental (% Variables)	CPLEX	CBC		
ARAI	44	0.01	0.01 (39.5%)	0.12 (12x, 12x)	1.18 (118x, 118x)	8	9
PR	52	0.02	0.01 (31.3%)	0.86 (43x, 86x)	3.70 (185x, 370x)	12	14
WANG	54	0.01	0.01 (8.29%)	0.86 (86x, 86x)	12.2 (1220x, 1220x)	12	14
LEE	58	0.01	0.01 (3.02%)	0.26 (26x, 26x)	2.88 (288x, 288x)	12	14
MCM	74	0.54	0.34 (10.4%)	6.19 (11x, 18x)	24.6 (46x, 72x)	15	16
DIR	76	0.14	0.01 (6.18%)	1.51 (11x, 151x)	11.5 (82x, 1550x)	14	15
HONDA	105	0.02	0.02 (0.95%)	9.06 (453x, 453x)	104 (5200x, 5200x)	27	33
CHEM	349	TO	1.42 (0.12%)	TO (n/a, n/a)	TO (n/a, n/a)	85	89
U5ML	857	0.01	0.01 (0.00%)	20.8 (2080x, 2080x)	TO (n/a, n/a)	261	264

cremental mode). The resulting set of SDC constraints are more likely to be consistent, making it easier for SDC to return a feasible solution after fewer iterations in propagation. Table 2.2 shows that SDS scheduler running in incremental mode remains scalable as the number of resources in the constraints increases.

Table 2.2: Runtimes in seconds for different combinations of resource constraints on multiplier and memory port — Results are shown for SDS scheduling in incremental mode.

Benchmark	# Operations	Runtime for Incremental Scheduling (sec)				
		1 mult 1 port	2 mult 2 port	3 mult 3 port	4 mult 4 port	6 mult 6 port
ARAI	44	0.01	0.01	0.01	0.01	0.01
PR	52	0.01	0.01	0.01	0.01	0.02
WANG	54	0.01	0.01	0.01	0.01	0.02
LEE	58	0.01	0.01	0.01	0.01	0.02
MCM	74	0.05	0.34	0.01	0.13	0.07
DIR	76	0.02	0.01	0.01	0.01	0.01
HONDA	105	0.01	0.03	0.04	0.09	0.24
CHEM	349	1.49	1.42	1.10	2.92	4.33
U5ML	857	0.01	0.01	0.01	0.01	0.01

To demonstrate the applicability of SDS, I further integrate SDS scheduler into LegUp [CCA+11], a state-of-the-art open-source HLS tool. I leverage LegUp's front-end to compile the input program into a CDFG and extract the relevant scheduling (e.g., timing, resource) constraints. SDS scheduler schedules the CDFG based on the constraints extracted from LegUp and returns the generated schedule to LegUp for post-scheduling processing and RTL generation. For experiments, I synthesize a set of applications from the CHStone benchmark suite [HTH+08] targeting the Intel Cyclone V FPGA at a clock period of `10ns`.

Using LegUp, we can compare the QoR of the synthesized hardware produced by SDC-based scheduling against the QoR of hardware produced by the SDS scheduler. For each benchmark, Table 2.3 reports the total number of operations of the program, runtime of SDS scheduling, as well as the key quality metrics post place-and-route generated by SDC-based scheduling and the SDS scheduler. Table 2.3 shows that the SDS scheduling approach achieves QoR comparable to that of SDC-based scheduling. On average, we observe small increase in clock period with small reduction in resource usage. Because

Table 2.3: Experiments on synthesizing CHStone benchmarks targeting the Intel Cyclone V FPGA at a clock period of `10ns` — `#Ops`: number of operations in the program. `#States`: number of states in the generated schedule for each function; benchmarks achieving state reduction with SDS are highlighted in bold. `CP`: achieved clock period in `ns`. `ALM`, `LUT`, `FF`, `DSP`, and `RAM`: number of corresponding resources used on the target device. `Runtime`: time in seconds taken to solve the SDS scheduling problem.

Benchmark	#Ops	SDC-Based Scheduling							SDS Scheduling							
		#States	CP	ALM	LUT	FF	DSP	RAM	Runtime	#States	CP	ALM	LUT	FF	DSP	RAM
ADPCM	850	25, 58	11.2	5316	8948	9851	122	7	0.03	25, **54**	12.5	5549	9166	9894	146	7
AES	812	37, 25, 17, 46	10.6	5313	7817	9568	0	10	0.05	37, 25, 17, **42**	11.5	5506	8147	9755	0	10
BLOWFISH	687	74, 36	7.5	2209	3330	4035	0	29	0.02	**70**, 36	7.8	2402	3709	4582	0	29
DFADD	361	4, 4	7.3	1439	1770	2124	0	1	0.01	4, 4	7.4	1442	1778	2105	0	1
DFDIV	361	65	9.7	3179	4383	6776	48	2	0.01	65	9.6	3170	4385	6769	48	2
DFMUL	279	5	9.6	1125	1601	1494	32	1	0.01	5	9.7	1126	1630	1492	32	1
DFSIN	1067	4, 9, 65	10.2	8584	10677	14568	82	5	0.05	4, 9, 65	9.6	8541	10594	14557	82	5
GSM	966	7	10.2	3256	4747	5204	54	7	0.03	7	10.6	3233	4697	5154	62	7
JPEG	2255	36, 9, 6, 7, 9, 7, 53	13.4	17066	28087	21211	87	83	0.11	36, 9, 6, 7, 9, 7, 53	13.7	17020	28112	21016	87	83
MIPS	346	5	11.7	1036	1468	928	6	4	0.01	5	11.8	1029	1468	947	6	4
MOTION	284	4, 7	8.4	5577	8257	8495	0	6	0.01	4, 7	9.0	5729	8339	8985	0	6
SHA	314	11, 11	6.1	1350	1596	2650	0	20	0.01	11, 11	6.3	1375	1603	2687	0	20

most of the CHStone benchmarks are not dominated by resource constraints, they do not benefit from reduction in the number of states with the exception of ADPCM, AES, and BLOWFISH. Nevertheless, these experiments demonstrate that the SDS scheduling approach is practical for real-life applications of non-trivial size. Note that the achieved clock period exceeds the target clock period for several benchmarks regardless of the scheduling approach applied. I believe this is a result of inaccurate delay estimation in HLS tools instead of an artifact of our proposed scheduling approach. Table 2.4 demonstrates that ADPCM, AES, BLOWFISH, and DFMUL can achieve further state reduction after we tighten the resource constraints in LegUp to one memory port and one multiplier.

Table 2.4: Benchmarks achieving further state reduction after tightening resource constraints — Results are shown for one memory port and one multiplier. Same notations are followed as in Table 2.3.

Benchmark	SDC-Based Scheduling		SDS Scheduling		
	#States	CP	Runtime	#States	CP
ADPCM	31, 64	12.0	0.04	**26**, **60**	11.3
AES	37, 49, 33, 55	10.4	0.05	37, 49, 33, **47**	10.5
BLOWFISH	118, 57	8.2	0.02	**108**, 57	8.5
DFMUL	7	9.4	0.02	**6**	9.6

2.5 Related Work and Discussions

Resource-constrained scheduling has been the subject of extensive study, resulting in a line of heuristics, including Hu's Algorithm, List Scheduling, and Force-Directed Scheduling, to solve the problem efficiently. Iterative meta-heuristics, such as simulated annealing and ant colony optimization, have also been demonstrated as viable options [De 94]. Because resource-constrained scheduling maps naturally to a constraint satisfaction problem consisting of logical connectives of linear constraints, it can also be solved with modern SMT solvers, which integrate specialized (linear) solvers with propositional satisfiability search techniques to achieve conflict-driven learning [DMB11]. In particular, a subset of SMT solvers focus on determining the satisfiability of a Boolean combination of difference constraints [WIGG05]. These solvers take advantage of an graph-based algorithm to efficiently explore the search space.

SDS scheduler is inspired by the concept of SMT and employs a graph-based algorithm to perform conflict-based learning to quickly prune out the infeasible search space. However, unlike generic SMT solvers in which SAT assumes a principal role in driving the underlying theory solver, SDS treats SAT and the underlying theory as equal partners. Notably, SDS' underlying theory is able to influence the subset of SAT clauses that need to be included at each iteration of the feedback loop and determine the appropriate problem that needs to be solved by SAT. In addition, SDS makes heavy use of well-established heuristics specific to the resource-constrained scheduling problem to significantly improve the efficiency of propagation. These problem-specific knowledge provides supports for the key features of the solver, including optimization, resource-aware lower bounding, and incremental learning described in Section 2.3.

Branch-and-bound style pruning is another popular approach for solving the resource-constrained scheduling problem [NR01, CHPM13]. This type of approach divides the problem into sub-problems and computes the lower and upper bounds of each sub-problem. A sub-problem is solved optimally when the lower and upper bounds coincide. While these branch-and-bound style schedulers employ problem-specific knowledge from lower and upper bounding to reduce overall scheduling time, SDS applies conflict-driven learning tightly coupled with various scheduling heuristics (including upper and lower bounding) to achieve additional runtime improvement. SDS' approach combines the power of conflict-driven learning and problem-specific knowledge to realize significant speedup. While previous schedulers are designed to work with only resource-constrained scheduling problems, the proposed joint SDC and SAT formulation allows more expressive encoding of a rich set of constraints. With a combination of SAT and SDC, SDS provides the flexibility to make tradeoffs among different constraints and select the encoding most suitable for each type of constraints.

Given the state-of-the-art with SDC, SDS aims to push the boundary of what is practically scalable and redefine the frontier of quality vs. scalability tradeoff. While this work focuses on HLS, the proposed scheduling approach can equally apply to resource-constrained scheduling problems in many other fields of study. Moreover, the scheduling framework is designed to generalize to a wide range of constrained scheduling problems with a variety of constraints. In Chapter 3, for example, we will describe how to

extend this framework to pipeline scheduling [ZL13, CBA14], which are also typically handled by heuristics for efficiency. In addition, recent interest in dynamically scheduled HLS [TLZ+15, LBC15, DZL+17, LTD+17, JBI17] necessitates a tradeoff between runtime hardware overhead and performance that may not be easily optimized. A scheduling formulation with SAT will enable modeling of the hardware resource overhead so it can be co-optimized during scheduling. The proposed scheduling approach can also be extended to handle cross-layer HLS optimizations, such as mapping-aware scheduling [TDGZ15, ZTDZ15] and place-and-route aware HLS [ZGRC14], as well as low-power optimizations in HLS [JZPC08, ZCDC15]. Because many constraints cannot be anticipated by heuristics, the gap to optimality is expected to only widen. Efforts in exact scheduling is therefore crucial for handling a rich set of current and future constraints.

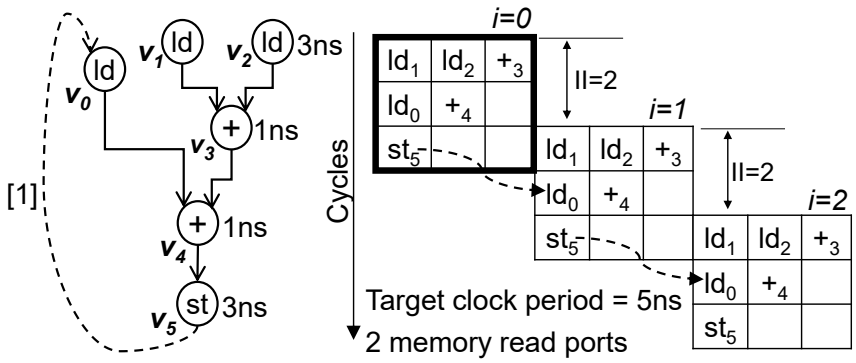
CHAPTER 3
EXACT MODULO SCHEDULING WITH JOINT SDC AND SAT

As loops abound in high-level software programs, loop pipelining is an important optimization in HLS because it allows different iterations of a loop to be overlapped during execution in a pipelined parallel fashion. Typically enabled by modulo scheduling [CLG02], loop pipelining creates a static schedule for a single loop iteration so that the same schedule can be repeated at a constant *initiation interval (II)*. Because II dictates the achieved throughput of the pipeline, minimizing the II is considered the foremost objective of pipeline scheduling.

While II determines the amount of parallelism, it is inherently limited by inter-iteration dependence (i.e., *recurrence*) between operations in different loop iterations. Figure 3.1(a) shows the DFG of a loop that we will referred to throughout this chapter. A static schedule for a single iteration is shown in bold in Figure 3.1(b). Due to the inter-iteration load-after-store dependence (indicated by the dashed arrow in Figure 3.1(a)) between v_5 and v_0, a subsequent iteration must start at least two cycles after the current iteration as shown in Figure 3.1(b). Any shorter II causes a dependence violation.

In addition to recurrence, II is also constrained by the available number of resources. Because the schedule for different loop iterations overlap in time, sufficient resources must be allocated to enable parallel execution of operations across iterations. As shown in Figure 3.1(b), the pipeline execution with `II=2` requires at least two memory read ports. If the same schedule targets `II=1`, at least three read ports are required due to the overlap among load operations.

Because modulo scheduling is not trivial in the presence of both recurrence and resource constraints, there exists a set of heuristics to efficiently solve the problem. For example, iterative modulo scheduling [Rau94] applies a list scheduling like heuristic with backtracking and has been adapted for loop pipelining in HLS [CCA+11]. However, state-of-the-art HLS tools typically employ the more versatile heuristic based on SDC (introduced in Chapter 2) to naturally handle operation chaining and various hardware-specific constraints [ZL13, CBA14]. SDC-based modulo scheduling is rooted in a linear programming formulation and can globally optimize over constraints that can be represented in



(a) (b)

Figure 3.1: An example modulo scheduling problem — *(a)* DFG of the loop to be scheduled. *(b)* Static schedule for a single iteration is shown in bold. Pipeline executes by repeating the same static schedule with an II of 2 cycles.

the integer difference form, including both intra- and inter-iteration dependences. Notably however, resource constraints cannot be exactly modeled within an SDC formulation. As a result, SDC-based modulo scheduling resorts to incremental scheduling of resource-constrained operations on top of SDC to heuristically legalize the schedule under resource constraints. Because resource constraints are not handled exactly, SDC-based modulo scheduling lacks guarantee on achieving the optimal II.

To address this problem, this chapter proposes a modulo scheduling formulation that couples SDC with SAT to exactly handle both timing and resource constraints for HLS pipelining. Similar to unpipelined scheduling with joint SDC and SAT [DLZ18] described in Chapter 2, the proposed approach exploits the efficiency of SDC while leveraging the scalability of SAT to quickly prune away infeasible schedule space and derive an optimal modulo schedule. However, modulo scheduling requires a modified SDC and SAT formulation (Section 3.2) and calls for a different problem-specific specialization technique (Section 3.3). The specific contributions of this chapter are as follows:

1. I propose a joint SDC and SAT formulation to exactly encode both resource and timing constraints for HLS pipelining.

2. I develop an optimal algorithm based on conflict-driven learning to efficiently solve the modulo scheduling problem.

3. I leverage problem-specific specialization to reduce the problem size and further achieve improved scalability.

The rest of this chapter is organized as follows: Section 3.1 provides background on pipeline scheduling and relevant theories; Section 3.2 details the proposed modulo scheduling formulation and conflict-driven scheduler; Section 3.3 describes specialization technique for the scheduler; Section 3.4 presents experimental results; Section 3.6 discusses related work.

3.1 Preliminaries

A typical HLS tool employs a software compiler (e.g., LLVM, GCC) to compile the input software program into a CDFG. Within this CDFG, subgraphs corresponding to loops to be pipelined are extracted for modulo scheduling, while the rest are synthesized with unpipelined scheduling techniques [CLN+11]. In this chapter, we focus on the following HLS modulo scheduling problem:

Given: *(1)* A loop represented by a CDFG with intra- and inter-iteration dependences. *(2)* A set of scheduling constraints which may include resource, latency, and relative timing constraints.

Objective: Generate a modulo schedule that minimizes the II while satisfying all given constraints.

Each operation in the CDFG is associated with a value that indicates the combinational delay of the operation. These delays are used to chain operations into the same cycle based on the target clock period of the problem. In Figure 3.1(a), we label the combinational delays for the four distinct types of operations in the graph. These delays are used during scheduling to satisfy the clock period constraint denoted in Figure 3.1(b). Unlike intra-iteration dependence edge, each inter-iteration dependence edge is associated with a distance indicating the number of loop iterations between the occurrences of the dependent operations. In Figure 3.1(a), the inter-iteration dependence edge in dash has a distance of 1 indicating that v_0 of the next iteration depends on v_5 of the current iteration.

In addition, the problem consists of resource constraints in the form of a resource model containing a set of different resource types (e.g., memory port, floating point multiplier). There exist a finite number of resources of each type in the resource model. If an operation requires any resource from the resource model to execute, we call this operation a *resource-constrained* operation. For example, the schedule in Figure 3.1(b) is derived based on a resource constraint of two memory read ports (as indicated), which allows at most two load operations in each cycle.

A modulo scheduling solution assigns each operation i to a *time step* t_i which indicates the cycle at which the operation executes. Due to the modulo nature of the scheduling from overlapping different iterations, each time step t_i corresponds to a (modulo) *time slot* s_i, where $s_i = t_i \% II$. For the bolded schedule in Figure 3.1(b), store operation v_5 is scheduled in time step 2 with no other operations. However, it is assigned to time slot 0 along with loads from v_1 and v_2 and addition from v_3. While there can be as many time steps as needed, there can be at most `II` time slots. A modulo reservation table (MRT) indicates the number of each type of resource used by all operations scheduled in each time slot. A feasible modulo scheduling solution requires an MRT in which no resource is oversubscribed in any time slot.

Because modulo scheduling is generally NP-hard under both resource and recurrence constraints, many heuristics have been proposed to quickly derive a solution with a small II [RG81, Lam88]. Among various heuristics, iterative modulo scheduling leverages backtracking to achieve better II [Rau94]. However, it provides no guarantee on attaining the optimal II, just like all other heuristics. To obtain optimal II, there exist enumeration-based approaches to exactly solve the problem [AG98]. Given the state of the field, we focus on describing the best known heuristic and exact modulo scheduling techniques in Sections 3.1.1 and 3.1.2.

3.1.1 SDC-based Formulation

In general, SDC-based scheduling [CZ06] declares a variable t_i to denote the clock cycle (time step) at which operation i in the CDFG is scheduled. Timing constraints, such as dependence and cycle time constraints, can then be represented exactly as the differences of these variables. To handle data dependence for modulo scheduling in particular, SDC

creates the following difference constraint

$$t_i - t_j \leq II \cdot Dist_{ij} - L_{ij} \tag{3.1}$$

where L_{ij} is the minimum latency between operation i and j, and $Dist_{ij}$ is the distance of the dependence. To schedule the DFG in Figure 3.1(a), for example, we impose the constraint $t_0 - t_4 \leq 0$ to honor the intra-iteration dependence between v_0 and v_4. This ensures that v_4 is scheduled no earlier than v_0. Note that an intra-iteration dependence corresponds to a dependence distance $Dist_{ij} = 0$ in the constraint in Eq. (3.1). Similar constraints are constructed for other intra-iteration data dependence edges. For the inter-iteration dependence in Figure 3.1(a), we impose the constraint $t_5 - t_0 \leq II - 1$, where II is the II currently being targeted. Intuitively, this constraint imposes a deadline for the schedule time of v_5 relative to the schedule time of v_0 beyond which v_5 from the current iteration will not execute in-time to produce results needed by v_0 of the following iteration.

To honor the target clock period T_{clk}, SDC identifies the maximum critical combinational delay $D(ccp(v_i, v_j))$ between pairs of operations v_i and v_j and impose the difference constraint in Eq. (3.2) to ensure pairs of operations whose critical delay exceeds the target clock period are scheduled in different cycles.

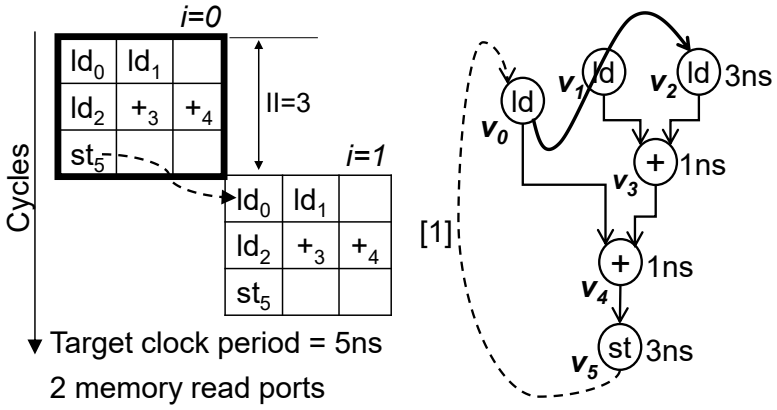
$$t_i - t_j \leq -1 \;\; \forall \, (v_i, v_j) \ni D(ccp(v_i, v_j)) > T_{clk} \tag{3.2}$$

For our example in Figure 3.1(a), we impose $t_2 - t_5 \leq -1$ to separate $v2$ and $v5$ into different cycles because the critical combinational path from $v2$ to $v5$ exceeds the target clock period of `5ns`. Similar constraints are imposed between v_0 and v_5 as well as v_1 and v_5.

While timing constraints can be handled naturally in SDC, resource constraints are difficult to represent even heuristically because the non-linearity of the MRT requires that operations using the same resource must not only be scheduled in different time steps but also in different time slots. As a result, simple partial ordering constraints in the form of $t_i - t_j \leq -1$ used to produce resource-abiding schedules in unpipelined SDC scheduling [CZ06] fail to honor the complete set of resource constraints in the case of modulo scheduling. To handle resource constraints, SDC-based modulo scheduling [ZL13, CBA14] rely on stepwise legalization of the non-resource-constrained SDC

schedule against the MRT to heuristically derive a solution, resulting in no guarantee on optimality.

For example, Zhang and Liu [ZL13] greedily commits operations from the non-resource-constrained SDC schedule. For each operation encountering resource conflict, a new SDC constraint is imposed to delay the operation, after which the SDC solution is updated. The II is increased if the SDC becomes infeasible at any point. This approach is fast but sub-optimal, as illustrated in Figure 3.2(a), which presents a possible schedule generated by such heuristic for our DFG in Figure 3.1(a). Because the load for v_0 is heuristically scheduled in the first instead of the second time slot, II must be increased to three cycles to accommodate the inter-iteration dependence (dashed arrow). Scheduling v_0 in the first time slot pushes either v_1 or v_2 to the second time slot due to the resource constraint of only two read ports. In either case, the length of the recurrence is increased, as shown in Figure 3.1(b). Canis et al. [CBA14] further enables backtracking to invalidate infeasible heuristic ordering in an attempt to achieve better quality at the cost of additional runtime. However, this technique requires tuning knobs whose value depends on the particular design and still provides no guarantee on optimality.



(a) (b)

Figure 3.2: Heuristic ordering of resource-constrained operations — *(a)* Achieved II is sub-optimal due to heuristic ordering of dependent load and store operations. *(b)* Postponing v_2 increases length of the recurrence and thus the II.

3.1.2 ILP-based Formulation

In place of heuristics, ILP can be applied to exactly model the modulo scheduling problem. Eichenberger and Davidson [ED97] leverages binary variable $a_{i,r}$ to indicate whether operation i is scheduled in time slot r in order to encode the modulo schedule. Timing and resource can then be constrained with this variable. Oppermann et al. [OKROS16] improves upon the resource handling capability of Eichenberger and Davidson by using binary variables to represent resource and time slot overlap instead of the actual modulo schedule. In particular, they propose to use binary overlap variable ϵ_{ij} to denote whether operation i's resource instance index is strictly less than j's index, and μ_{ij} to denote whether operation i is executed in a time slot strictly earlier than the time slot of j. These binary variables are in turn constrained with resource index variable r_i, which denotes the index of resource instance used by operation i, and time slot variable s_i, which denotes the modulo time slot of operation i, as in Eichenberger and Davidson. Given these constraints on the consistency of variables ϵ_{ij}, μ_{ij}, r_i, and s_i, resource constraints are satisfied by ensuring that every pair of operations (i, j) use difference resource instances, or are scheduled at different time slots as followed:

$$\epsilon_{ij} + \epsilon_{ji} + \mu_{ij} + \mu_{ji} \geq 1 \tag{3.3}$$

Even though the ILP formulations handle all constraints exactly and can return a schedule that satisfies a specific II given enough time, ILP is in general NP-hard and difficult to scale. ILP also requires significantly more variables than SDC for encoding the same problem and is too general to exploit problem-specific properties.

3.2 Modulo Scheduling with Joint SDC and SAT

Given the tradeoff between scalability and quality in the comparison between SDC and ILP in Section 3.1, I propose a modulo scheduling algorithm that integrates SDC and SAT to exactly handle various types of constraints and optimally solve the modulo scheduling problem. Borrowing the idea of unpipelined scheduling with joint SDC and SAT [DLZ18], the proposed formulation leverages SDC to naturally handle timing constraints and SAT to exactly encode resource constraints.

As shown in the high-level diagram in Figure 3.3, the proposed modulo scheduler is composed of a conflict-based SAT solver coupled with a graph-based SDC solver in a conflict-driven learning loop. On the left, we have a SAT solver that takes advantage of conflict-based search (detailed in Section 3.2.1) to propose, what we referred to as, modulo orderings that satisfy resource constraints imposed by the MRT. These modulo orderings are converted into difference constraints in SDC and inserted into the SDC problem. On the right, we have an SDC solver that takes advantage of graph-based traversal (detailed in Section 3.2.2) to check the feasibility of the modulo orderings. Any infeasibility is encoded as a conflict clause in SAT and added to the SAT problem. Given the practical scalability of SAT and the efficiency of SDC, our solver iterates between SAT and SDC (described in Section 3.2.3) until a feasible solution is found or proven to be non-existent.

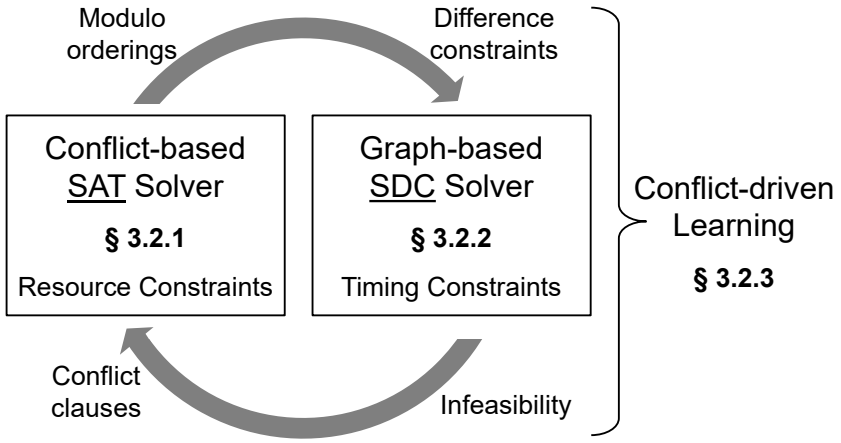


Figure 3.3: Overall structure of the proposed modulo scheduler.

3.2.1 Resource Constraints in SAT

To handle resource constraints, we declare binding variable B_{ik} to denote whether operation i is bound to resource instance k. B_{ik} is true if operation i is bound to resource instance k. By constraining the binding variables with the SAT clause $\sum_k B_{ik} = 1 \; \forall i$, we can ensure that each resource-constrained operation is assigned to exactly one resource instance and that no resource is oversubscribed. With the binding variables, it follows that sharing variable R_{ij} can be derived to denote whether operation i is sharing the same

resource instance with operation j as followed:

$$R_{ij} = \bigvee_{k \in T_p} (B_{ik} \wedge B_{jk}) \tag{3.4}$$

where T_p denotes resource instances of type p. R_{ij} is true if both operations i and j are bound to the same resource instance.

Pipeline scheduling prohibits two operations i and j that share the same instance of resource from being scheduled in the same modulo time slot. In other words, $t_i - t_j \neq kII \ \forall \ k \in \mathbb{Z}$, which translate to a disjunctive set of constraints $kII < t_i - t_j < (k+1)II$ and $kII < t_j - t_i < (k+1)II$. Therefore, we introduce modulo ordering variables $O_{i \to j,k}$ to represent these constraints as follows:

$$O_{i \to j,k} = True \ \mapsto \ (k-1)II < s_i - s_j < kII \ \forall \ k \in \mathbb{Z} \tag{3.5}$$

$$O_{i \to j,k} = False \ \mapsto \ \emptyset \tag{3.6}$$

As shown in Eq. (3.5), assigning $O_{i \to j,k}$ to true maps to the difference constraint where operation i must be scheduled in an earlier cycle than j. Furthermore, their distance must be greater than kII and less than $(k+1)II$ cycles, which means that they are separated apart by k II-intervals. As shown in Eq. (3.6), assigning $O_{i \to j,k}$ to false maps to an empty set of constraints, indicating that it is not necessary to impose any partial ordering between operations i and j because the particular resource binding does not require any partial ordering. Note that k can be bounded by the length of any non-modulo schedule or the lengths of recurrence cycles. Here we use T to denote the bounded space of k. Ultimately, $O_{i \to j}$ is derived from modulo orderings as:

$$O_{i \to j} = \sum_{k \in T} O_{i \to j,k} \leq 1 \tag{3.7}$$

As shown in Eq. (3.7), $O_{i \to j}$ is true if operation i is scheduled before j, and they are not in the same time slot.

Given the mapping between SAT and SDC, the following clauses are included to connect the the sharing variables with the modulo ordering variables:

$$R_{ij} \to (O_{i \to j} \vee O_{j \to i}) \tag{3.8}$$

$$\neg (O_{i \to j} \wedge O_{j \to i}) \tag{3.9}$$

Eq. (3.8) indicates that if operations i and j share the same resource instance, operation i must be scheduled either in an earlier cycle or in a later cycle than operation j, but certainly not in the same time slot. Eq. (3.9) ensures that operation i cannot be simultaneously scheduled both in an earlier cycle and later cycle than j.

3.2.2 Timing Constraints in SDC

Timing constraints in SDC can be conveniently represented using a constraint graph in which each variable maps to a node and each constraint maps to an edge in the graph. Figure 3.4(a) shows the set of intra-iteration dependence, inter-iteration dependence, and cycle time constraints in SDC form for our example. These constraints map to the nodes and edges in the constraint graph in Figure 3.4(b). For each constraint in the integer difference form $t_i - t_j \leq C_{ij}$, the constraint graph includes an edge of weight C_{ij} from node j to i. For clarity, note that we have omitted the weights for zero-weight edges.

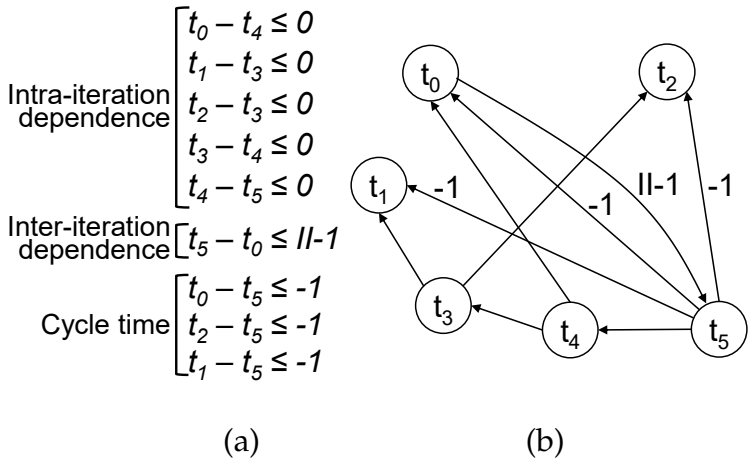


Figure 3.4: SDC constraints and corresponding SDC constraint graph — *(a)* Timing constraints in SDC. *(b)* SDC constraint graph.

With this graph-based representation, we can easily derive a feasible schedule, either ASAP or ALAP schedule, by solving a single source shortest path problem. In addition, we can conveniently detect infeasibility of the difference constraints by the presence of negative cycle in the graph. For example, adding the SDC constraint $t_0 - t_2 \leq -1$ to the system in Figure 3.4(a) induces the dashed edge from t_2 to t_0 in Figure 3.5, creating a negative cycle (shown in bold) that indicates the system is infeasible.

3.2.3 Conflict-Driven Learning

As shown in Figure 3.3, SAT and SDC work closely in a loop to handle both resource and timing constraints exactly and efficiently. For each iteration, SAT makes a proposal of modulo orderings that satisfy resource constraints in Eq. (3.4), (3.7), (3.8), and (3.9) by determining a satisfiable assignment for the modulo ordering variables $O_{i \rightarrow j,k}$. SDC constraints used to enforce resource constraints are created based on the mapping in Eq. (3.5) and appended to the SDC graph. SDC checks the feasibility of the updated graph and returns any feasible solution as the schedule if the graph is feasible. If the graph is infeasible, SDC instead returns the modulo ordering edges involved in the negative cycle that causes the infeasibility. The infeasibility is encoded as conflict clause and appended to the SAT problem. In future iterations, SAT will no longer propose any modulo ordering that violates any previously added conflicts. The solver iterates until a feasible solution is found or the SAT search space is exhausted. While there may be multiple feasible solutions, our solver returns the earliest encountered feasible schedule.

Figure 3.5 illustrates a single iteration of the conflict-driven learning process with our target `II=2`. In this iteration, assume that SAT proposes a modulo ordering that assigns variable $O_{0 \rightarrow 2,0}$ to true, which enforces that operation 0 is scheduled before 2, and that the two operations are less than one II apart. The two corresponding SDC constraints are shown on the left. SDC then adds the edges (in dash) corresponding to these constraints to the graph and detects a negative cycle (in bold) that involves one of the two newly added edges. The involvement of this edge in the negative cycle results in the conflict clause on the left, which prevents any modulo ordering with operation 0 scheduled before 2 and within one II apart. SAT will then continue onto the next iteration with a different proposal that satisfies resource constraints but does not involve this previously infeasible edge. As you can see, SAT learns from previously infeasible edges in each iteration to prune the search space. We leverage negative cycle to keep the generated SAT conflict clauses as short as possible because shorter conflicts are able to prune out more of the search space in SAT and result in faster convergence of the solver. If the search space is completely pruned out before finding a feasible solution, the problem is infeasible for the particular II. Otherwise, the problem is feasible where the shortest path solution serves as the feasible schedule.

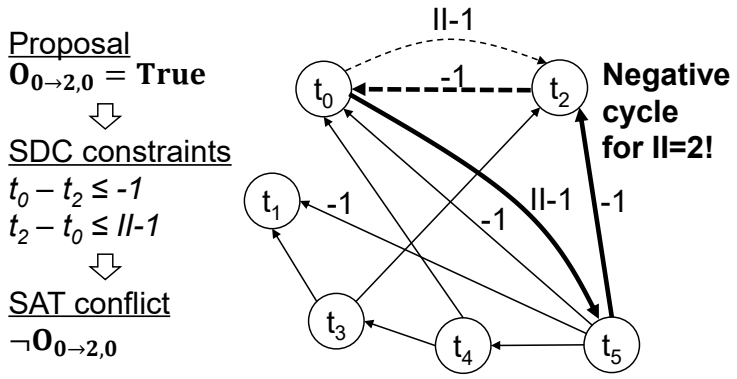


Figure 3.5: One iteration of conflict-driven learning.

3.2.4 Optimization

While we take advantage of conflict-driven learning to derive a modulo schedule that satisfies a particular II, we optimize for the best II by starting with a lower bound value for the II and incrementing it one at a time until the II is feasible or an upper bound value has been reached. Because this is a conventional optimization technique typically employed by HLS tools [CLN+11, CCA+11], the proposed conflict-driven learning algorithm is generally applicable regardless of the approach used to establish such upper and lower bounds on II.

3.3 Graph-based Problem Reduction

In general, the graph to be modulo scheduled can be partitioned into acyclic and cyclic subgraphs. The acyclic subgraphs contain only forward edges, while the cyclic subgraphs contain backward edges in addition forward edges. Because resource constraints essentially delay the execution of resource-constrained operations, resource constraints may cause violation with timing constraints imposed by backward edges. Due to the interaction between backward edges and resource constraints, exactly scheduling the cyclic subgraphs in the presence of resource constraints constitutes the "hard" aspect of the modulo scheduling problem. Subgraphs that are acyclic or not constrained by resource can be scheduled efficiently and exactly with heuristics. Based on this observation, I propose to further accelerate the performance of exact modulo scheduler by reducing the

complexity of the exact modulo scheduling problem to that of exactly scheduling the cyclic resource-constrained subgraphs.

To enable graph reduction, we rely on the strongly connected component (SCC) algorithm [Tar72] to partition the graph into cyclic and acyclic components. This results in a directed acyclic graph (DAG) of the SCCs of the input graph. Some SCCs form a *trivial subgraph* consisting of only a single node, while others form a *non-trivial subgraph* consisting of multiple nodes. Those with multiple nodes must be cyclic. We refer to cyclic SCCs with one or more resource-constrained nodes as *complex subgraphs* and cyclic SCCs with no resource-constrained nodes as *basic subgraphs*. In Figure 3.6(a), component C is a complex subgraph because it contains a resource-constrained memory operation. Otherwise, it would have been categorized as a basic subgraph. Because they are single nodes, v_1, v_2, and v_3 each constitutes a trivial subgraph regardless of whether they are resource-constrained.

We combine all basic subgraphs without connections between them into a *basic supergraph*. This basic supergraph can be solved exactly with SDC-based modulo scheduling because there is no resource constraint. The scheduling solution of this basic supergraph will satisfy all timing constraints imposed by the edges within the supergraph. Similarly, we combine all complex subgraphs without connections in between into a *complex supergraph*. However, due to the interaction between timing constraints (from backward edges) and resource constraints, this complex supergraph must be solved with an exact technique such as the proposed joint SDC and SAT modulo scheduling algorithm detailed in Section 3.2. The solution of the complex supergraph will satisfy all timing constraints imposed by edges in the supergraph as well as the resource constraints of the modulo scheduling problem. Regardless of basic or complex supergraph, the schedule generated serves as a *relative schedule* that we can use later to commit the final schedule. Operations satisfy the relative schedule as long as the relative time positions at which operations execute remain unchanged. For example, the relative schedule in Figure 3.6(a) is satisfied as long as the store and addition operations are executed one cycle after the load operation.

To motivate the subsequent procedure in Algorithm 1 for committing the final schedule based on the relative schedules of basic and complex supergraphs, we identify several interesting properties of the different types of subgraphs. For better intuition, we provide

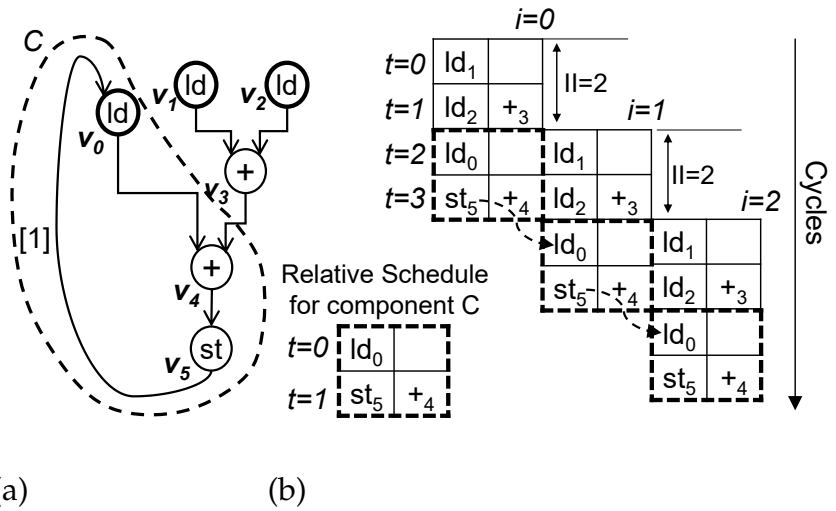


Figure 3.6: Illustration of graph-based problem reduction — *(a)* C is a complex component that is scheduled first. *(b)* Final committed schedule.

an informal proof of each property to illustrate the main ideas. Here $t_G(i)$ denotes the schedule time of operation i in the relative schedule, and $t(i)$ denotes the schedule time in the committed schedule.

Property 1. *Given any time step T and a relative schedule $\{t_G(i), \forall i\}$ of a basic subgraph G, committing every operation i to time step $t(i) = T + t_G(i)$ satisfies all timing constraints imposed by G.*

The relative time positions between operations remain the same in the committed schedule $\{t(i), \forall i\}$ as in the original relative schedule $\{t_G(i), \forall i\}$ for G. Because the original relative schedule satisfies all timing constraints imposed by G, the committed schedule must also satisfy all timing constraints imposed by G.

Property 2. *Given any time step T and a relative schedule $\{t_G(i), \forall i\}$ of a complex subgraph G, there exists an integer constant $\delta : 0 \leq \delta < II$ such that committing every operation i to time step $t(i) = T + t_G(i) + \delta$ satisfies all timing constraints of the subgraph as well as the time slot assignment imposed by the relative schedule.*

Assume that the operations are first committed as in Property 1. If the time slot assignments are satisfied, we have obtained a committed schedule that satisfies both timing constraints and time slot assignments. If the time slot assignments are not satisfied, we

can repeatedly increase the time steps of all operations simultaneously by one cycle until the time slot assignments are satisfied. Due to the modulo nature of the schedule, we must be able to find a schedule that satisfies the time slot assignments within `II` cycles. Any schedule we commit also satisfies the timing constraints of the subgraph because we increase the time steps of all operations by the same constant number of cycles δ and maintain their relative positions in time.

For example, if we would like to commit the relative schedule in Figure 3.6(a) for $T = 1$, the load for v_0 will be committed to $t(0) = 1$ if $\delta = 0$. This commits v_0 to time slot 1, violating v_0's being scheduled in time slot 0 in the relative schedule. Therefore, the entire relative schedule must be delayed by one cycle to accommodate the time slot assignments. With $\delta = 1$, v_0, v_5, and v_4 will be committed to $t(0) = 2$, $t(5) = 3$, and $t(4) = 3$, where both timing constraints between these operations and their time slot assignments are satisfied.

Property 3. *Given any time step T and that the complex supergraph of the problem has been scheduled and committed to the MRT, the single operation i in each trivial subgraph G can always be committed at some time step $t(i) : T \leq t(i) < T + II$ without violating resource constraints.*

A trivial subgraph contains a single operation. If the operation is not constrained by resource, committing it at any time step will not violate resource constraints. If the operation is constrained by resource, there must be some slot with available resource in the MRT for the operation to be scheduled because enough II slots should have been pre-allocated to satisfy resource constraints. Because a single resource-constrained operation can be scheduled in any time slot in the MRT, committing it will not violate any resource constraints.

In Figure 3.6, assume v_0, v_5, and v_4 have been committed to slots 0, 1, and 1, respectively, after relatively scheduling the complex subgraph C. Further assume that v_1 is then schedule to $t(1) = 0$, which corresponds to slot 0. With these operations committed, all read port resources in slot 0 of the MRT are subscribed. Because of this, v_2 must be committed to $t(2) = 1$ and slot 1 because slot 0's read ports have been fully subscribed. However, there must be available resource in slot 1 for v_2 because the minimum resource-constrained II of 2 requires at least two modulo time slots in the MRT, each with two read ports available.

Algorithm 1 `ExactModuloScheduling(II)`

```
1: Partition the graph into its SCCs
2: Generate relative schedule for combined trivial subgraphs
3: with SDC-based modulo scheduling
4: Generate relative schedule for combined non-trivial subgraphs
5: with joint SDC and SAT modulo scheduling
6: Update MRT
7: for each component in topologically sorted order of SCCs do
8:    if component is a trivial subgraph then
9:       Schedule ASAP based on relative schedule
10:   else if component is non-trivial subgraph then
11:      Schedule ASAP based on relative schedule while satisfying
12:      time slot assignment
13:   else if component is single resource-constrained node then
14:      Schedule ASAP at time slot with available resource
15:      Update MRT
16:   else
17:      Schedule ASAP
18:   end if
19: end for
20: if Any above step is infeasible then II is infeasible end if
```

Based on the above properties, any operation i in subgraphs (regardless of type) can always be scheduled at some time step $t(i) \geq T$, given a reference time T, without violating timing constraints within the subgraphs or resource constraints of the modulo scheduling problem. Therefore, we can traverse the subgraphs (SCCs) in a topological order (because the graph of SCCs form a DAG) and commit the operations in each SCC to the earliest possible time step (i.e., ASAP). Dependence between subgraphs (manifested by forward edges) determines the earliest time step for which operations in each subgraph can be scheduled. If we consider this earliest time step as T in the previous properties, operations in the subgraph can be committed based on those properties from this reference time step. Committing subgraphs in topological order ensures that timing constraints between subgraphs, all of which must be forward edges, are also fully satisfied. Our exact modulo scheduling algorithm with graph-based problem reduction is listed in Algorithm 1.

In the algorithm, Line 7 commits operations in a basic subgraph to the final schedule based on Property 1 to satisfy timing constraints, while Line 9 commits operations in a complex subgraph to the final schedule based on Property 2 to ensure that both timing

and resource constraints are honored. If the subgraph turns out to be a single resource-constrained node, Line 11 commits the operation, based on Property 3, to the earliest possible time step whose corresponding time slot has available resource remaining in the MRT. If the subgraph is a single node that is not constrained by resource, it can be scheduled ASAP as shown in Line 14. Our algorithm is essentially an ASAP scheduling scheme subject to resource availability and any prior relative assignment of time steps (of basic and complex subgraphs) and exact assignment of modulo time slots (of complex subgraphs). Figure 3.6(b) shows the committed schedule for scheduling the graph in Figure 3.6(a) with Algorithm 1. Note that the graph reduction technique is generally applicable to any exact modulo scheduling techniques, including ILP.

3.4 Experiments

The proposed modulo scheduler is implemented in C++, interfaced with the Lingeling SAT solver [Bie13]. It is executed on an Intel Xeon CPU running at 2.5GHz and evaluated on a set of 350 benchmark loops from popular HLS benchmark suites CHStone [HTH+08] and MachSuite [RAS+14]. For experiment purpose, we further classify the benchmarks into *trivial*, *easy*, and *challenging* benchmarks to better evaluate the benefit of the proposed approach. In particular, trivial benchmarks contain no complex component in the graph. Given the proposed graph-based problem reduction technique, these benchmarks do not require exact modulo scheduling to be solved optimally. Therefore, they are not included in our runtime evaluations to avoid skewing the results. For the other benchmarks, we compare the runtimes of the proposed joint SDC and SAT scheduler, with and without graph reduction, against those of state-of-the-art commercial ILP solver CPLEX [IBM17] running the best known ILP-based modulo scheduling formulation [OKROS16] (described in Section 3.1.2). For convenience, we use `ILP` and `SDS+` to refer to the two scheduling techniques, but qualify each technique with `Default` or `Reduced` to indicate whether graph reduction has been applied.

Figure 3.7 summarizes the runtime speedup of SDS+ on easy benchmarks, which contain complex subgraphs but can solved by Default ILP in less than one second. Each bar represents the runtime speedup against Default ILP for one benchmark. Benchmarks are

ordered by increasing Default ILP runtime. Each color represents the additional speedup achieved over the previous solver configuration. For example, black bar shows the runtime speedup of Reduced ILP over Default ILP, while black and light gray together show the speedup of Default SDS+ over Default ILP. Black, light gray, and gray together indicate the speedup of Reduced SDS+ over Default ILP. Figure 3.7 shows that SDS+, with or without reduction, is consistently more competitive than ILP, with or without reduction. While simply applying the graph-based reduction technique on ILP leads to some degree of speedup, applying SDS+ achieves speedup that grows faster as the difficulty of the problem increases. In addition, Reduced SDS+ can achieve more than one order of magnitude speedup from Default ILP for the most difficult cases in this plot.

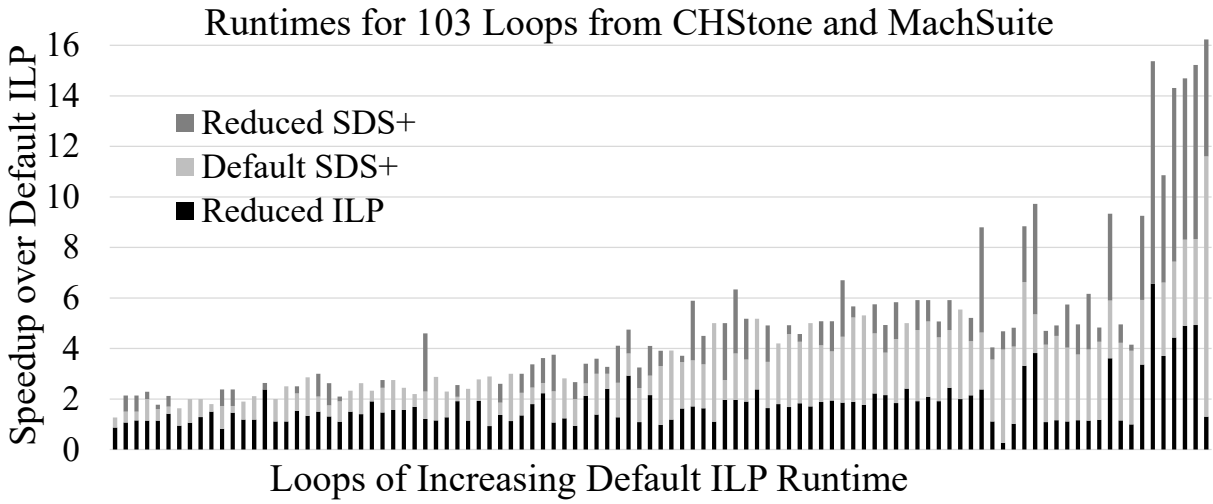


Figure 3.7: Runtime evaluation on easy loops — Default ILP's runtimes are less than one second. Each color represents the additional speedup achieved over the previous solver against Default ILP.

In Table 3.1, we evaluate the runtimes of more challenging loops, which contain complex subgraphs and require more than one second of ILP time. The loops are sorted by the total number of operations before reduction. However, the exact relationship between various metrics and speedup depends on the graph topology and the interaction between resource-constrained nodes and cyclic subgraphs. SDC and SAT complexity is determined by the number of operations in accordance to Section 3.2. Overall, while Reduced ILP provides marginal speedup from Default ILP, SDS+ (with and without reduction) is able to significantly widen the speedup gap for almost all the loops. With

graph reduction, Reduced SDS+ is especially competitive on the more difficult loops like `JPEG19` and `ADPCM2`, achieving over two orders of magnitude of speedup. For `JPEG23`, Reduced ILP's speedup is noticeable because graph reduction decreases the total number of operations by around 86% and the number of resource-constrained operations by 50%. As a result, SDS+ gives no further benefit on top of Reduced ILP in this case. On the other hand, Reduced ILP reaps negligible benefit for `JPEG87` because the graph contains a long recurrence cycle that encapsulates all resource-constrained operations. SDS+'s performance in this case shows that SDS+ is able to handle constraints more efficiently. Noticeably, Reduced SDS+ is able to solve `ADPCM1` and `DFSIN1` for which both ILP and Reduced ILP time out.

Benchmark	#Ops	Reduced #Ops	Default ILP	Reduced ILP	Default SDS+	Reduced SDS+
FFT_STRIDED2	84 / 18	24 / 18	4.29	1.11 (3.86x)	0.804 (5.33x)	0.091 (47.1x)
JPEG23	135 / 26	18 / 13	2.30	0.190 (12.1x)	0.294 (7.82x)	0.029 (79.3x)
JPEG18	175 / 23	114 / 23	8.60	3.03 (2.84x)	0.512 (16.7x)	0.529 (16.2x)
MD_GRID7	300 / 50	298 / 50	7.14	3.06 (2.33x)	0.285 (25.1x)	0.257 (27.8x)
JPEG87	380 / 37	370 / 37	7.13	6.71 (1.06x)	0.642 (11.1x)	0.361 (19.7x)
JPEG19	476 / 65	465 / 65	TO	397 (>2.27x)	3.41 (>264x)	2.35 (>383x)
JPEG17	942 / 93	615 / 68	156	96.1 (1.62x)	7.29 (21.4x)	6.93 (22.5x)
ADPCM2	710 / 114	295 / 80	TO	562 (>1.60x)	TO	2.01 (>448x)
ADPCM1	777 / 108	466 / 102	TO	TO	31.1 (>28.9x)	11.9 (>75.6x)
DFSIN1	2651 / 74	115 / 74	TO	TO	TO	606 (>10.0x)

Table 3.1: Runtime evaluation for more challenging loops — `#Ops` shows the total number of operations and number of resource-constrained operations before graph reduction. `Reduced #Ops` shows the same numbers after graph reduction. `TO` indicates time-out after 15 minutes or 10x SDS+ runtime, whichever is greater.

While runtime is not strictly proportional to node count, the ILP-based formulations become difficult to solve as the total number of operations increases beyond 400 or the number of resource-constrained nodes goes beyond 50. In these cases, SDS+ demonstrates improved scalability by combining the efficiency of SDC and SAT. Although the ILP formulation by Oppermann et al. [OKROS16] used in our experiments has demonstrated improved resource handling capability than previous formulations, it is still more susceptible to scalability issues because both timing and resource constraints are encoded as ILP constraints, which are NP-hard to solve. Instead, SDS+ handles the timing aspect

of the problem using polynomial-time SDC and leaves the resource aspect to SAT. In addition, our reduction technique can help prune out nodes to further alleviate scalability issues. For `JPEG19` and `ADPCM2`, graph reduction helps ILP become manageable.

3.5 Discussions

The primary objective of modulo scheduling is to find a schedule with the minimum II. As described in Section 3.2.4, HLS searches for the lowest II by attempting a series of IIs (starting from a lower bound) in a set of modulo scheduling problems. In each of these problems, if there is no additional objectives other than minimizing II, the solver is solving a feasibility problem that determines whether the program can be modulo scheduled with a particular II value.

With that said, one can additionally minimize the latency of the resulting modulo schedule as a secondary objective on top of the primary objective of minimizing II. In other words, one can find the ASAP (shortest-latency) modulo schedule with the lowest possible II given timing and resource constraints. For example, by using the sum of operations' schedule times as the objective, the ILP-based modulo scheduling formulations described in Section 3.1.2 can naturally optimize for latency to determine the ASAP schedule for a particular II. Although II will still be optimized outside the ILP solver, latency is handled within the solver itself. In contrast to the feasibility problem described previously, the solver in this case is instead solving an optimization problem that minimizes latency for a particular II value.

Given the distinction between the feasibility and optimization problem, it is necessary to realize that these problems introduce some sudden tradeoff involving runtime and applicability of a solver. In terms of runtime, for example, while it may be desirable to achieve the ASAP modulo schedule, the runtime overhead may overshadow the latency benefit because the optimization problem is in general much more difficult to solve than the feasibility problem. Intuitively, an optimization problem must prune out all feasible points that are sub-optimal, while a feasibility problem simply needs to locate only one out of possibly many feasible points. In terms of solver applicability, for example, while an ILP solver is designed to handle optimization problems, a SAT solver is designed to

address only feasibility problems. Despite SAT's practical scalability, if not performed correctly, it may not be necessarily efficient to adopt a SAT based approach to solve optimization problems.

As such, it is important to clarify that the proposed modulo scheduling approach, as described in this chapter, finds only a feasible minimum-II schedule, but provides no guarantee on the latency of such schedule. This means that it solves only the feasibility problem, not the optimization problem. In fact, the problem reduction technique proposed in Section 3.3 requires committing the final modulo schedule from relative schedules of subgraphs. The need to align relative schedules to modulo time slots in Algorithm 1 may introduce latency overhead depending on how the operations are scheduled relatively and the order in which subgraphs are committed to the final schedule. Based on our benchmarks, the latency reported by Reduced SDS+ is on average 1.3x of the optimal latency that can be achieved by Default ILP with optimization enabled. However, using Default ILP with optimization enabled incurs an average runtime which is 38x that of Reduced SDS+ solving the feasibility problem (without optimization). For future work, it would be interesting to explore this underlying tradeoff and to investigate techniques to reduce or minimize the latency overhead of the proposed problem reduction approach.

3.6 Related Work

Modulo scheduling has been traditionally applied in the domain of software pipelining to increase the amount of instruction-level parallelism by interleaving instructions among different iterations of a loop [RG81]. Because modulo scheduling is general NP-hard under both resource and recurrence constraints, many heuristics such as iterative modulo scheduling [Rau94] have been proposed to quickly derive a solution with small II while incurring small runtime overhead. In addition to the primary objective of minimizing II, a set of modulo scheduling heuristics have also been proposed to reduce register pressure for software pipelining [Huf93, LVAG95, LGAV96]. Other than heuristics, there are also a set of exact approaches for solving the modulo scheduling problem. In comparison to the ILP-based approach described in Section 3.1, Altman and Gao propose an enumeration-based approach to derive modulo schedules [AG98]. This approach avoids

the need to formulate the relevant timing and resource constraints into linear forms and works on DFGs with a small number of nodes. Eichenberger et al. also propose an ILP-based approach to minimize register pressure [EDA14].

CHAPTER 4
DYNAMIC HAZARD RESOLUTION FOR PIPELINING IRREGULAR LOOPS

As described in Chapter 3, conventional HLS pipelining typically leverages modulo scheduling [Rau94, Huf93, LGAV96], a compile-time optimization which creates a static schedule for a single loop iteration that can be repeated at a fixed `II`. The modulo scheduling algorithm analyzes the program's CDFG along with resource, data dependence, and other constraints to minimize the `II` while ensuring that the pipeline does not encounter hazards during execution. Specifically, the statically generated schedule must not allow multiple operations to access the same physical resource within a single cycle (*structural hazards*) and must ensure that dependences between memory loads and stores are not violated (*data hazards*). The need to avoid these two types of hazards on memory accesses often limit the throughput of the synthesized pipeline.

HLS pipelining makes extensive use of memory dependence and alias analysis to identify dependences and disambiguate memory accesses (for convenience, we use dependence and alias analysis interchangeably in subsequent discussions). Such techniques attempt to classify each pair of memory accesses as no-alias or must-alias if the analysis is conclusive, or may-alias if the analysis is inconclusive. Static alias analysis is able to return fairly accurate dependence information for programs with compile-time analyzable control flow and highly regular memory access patterns, allowing efficient pipeline schedules to be created. However, such static analysis techniques are ineffective against programs that contain conditional and/or data-dependent memory operations with memory addresses unknown at compile-time, making it difficult to prove the absence of aliases. As a result, the dependence information will be inexact and contains may-alias pairs that have to be treated as must-alias by the scheduler to ensure hazard-free execution under all circumstances.

While existing pipelining techniques are effective at generating high-throughput hardware for regular dataflow-centric applications with well-structured data access patterns, they cannot efficiently synthesize *irregular programs* (e.g. graph algorithms, data analytics, sparse matrix computations) because these programs exhibit data-dependent control flow, irregular memory dependence patterns, and dynamic workloads. In

particular, irregular programs incur structural and/or data hazards caused by conditional and/or data-dependent memory operations whose occurrence pattern cannot be accurately predicted by static compiler analysis, even with advances in polyhedral model [MDQ13,PZSC13]. To ensure functional correctness, the pipelining algorithm must conservatively assume that these hazards always occur, even if they rarely or never do in practice. Consequently, conservative static pipelining leads to pessimistic performance as the pipeline stalls needlessly to avoid hazards which may be *infrequent* during actual execution.

We illustrate this performance gap using Maximal Matching in Figure 4.1(a), a common graph algorithm that computes the set of independent edges without common vertices in a graph. The kernel examines the two endpoints of each edge of a graph and checks if they are marked. If not, the algorithm updates the vertices at the endpoints using two conditional stores. Note that there are conditional loop-carried dependences between the `load` operations on `line 5` and the `store` operations on `line 6/7`. However, these stores are executed infrequently for a dense graph because only a small subset of its edges are independent. Figure 4.1(b) shows the static schedule for one iteration of the loop. For this schedule, each array is mapped to a single-ported memory so only one memory access per array is allowed in each cycle. To avoid potential structural and data hazards, conventional techniques will pipeline this design to an `II` of 4 cycles, which results in a 17-cycle execution latency, as shown in Figure 4.2(a), for the first four iterations processing a fully-connected graph. Six cycles are unused because the condition on `line 5` in Figure 4.1(a) is evaluated false for all iterations except `j=0`, thus no stores need to be performed for those iterations.

To take advantage of the infrequent nature of conditional memory accesses and inter-iteration memory dependences in this case, a better solution would be to launch new iterations more frequently to increase the efficiency of the pipeline by saturating the available memory bandwidth. As shown in the ideal execution in Figure 4.2(b), aggressively launching a new iteration every two cycles from Cycle 4 onward reduces the execution latency to 13 cycles. However, aggressive pipelining causes structural hazards, when the `stores` from the current iteration collide with `loads` from the next iteration, as well as data hazards, when the loop-carried dependence is violated. For example, if the stores

```
for (j=0...num_edges){
 int s = e[j].src;
 int d = e[j].dst;

 if (v[s]<0 && v[d]<0){
  v[s] = d;
  v[d] = s;
}}
```

(a) Source code.

Cycle		
0	`e[j].load`	(line 2/3)
1	`v[s].load`	(line 5)
2	`v[d].load`	(line 5)
3	`v[s].store`	(line 6)
4	`v[d].store`	(line 7)

(b) Schedule for one iteration.

Figure 4.1: Maximal Matching example — (a) Source code in C-like syntax. (b) Static schedule produced by conventional HLS pipelining with `II=4`. Only `load` and `store` operations are shown while others (e.g., comparisons) are omitted.

to array `v` in iteration `j=1` were executed, they would collide with the loads from array `v` in iteration `j=2`. Moreover, a dependent load in iteration `j=2` may read from an address in array `v` before a store in iteration `j=1` writes to the same address, violating the inter-iteration read-after-write dependence between these memory accesses.

A possible approach to address the performance gap between conservative and aggressive pipelining is to *speculatively* execute each iteration, launching each iteration before hazard-free execution can be guaranteed, and rely on a hardware dynamic hazard resolution mechanism to resolve any hazard that actually occurs. To achieve high throughput using this approach, two problems must be addressed: first, aggressive pipelining must be performed without pessimistically assuming that conditional or data-dependent hazards always occur; second, hazards that actually occur must be detected and resolved appropriately at runtime.

To achieve this goal, I propose a set of synergistic techniques which enable dynamic hazard resolution in pipeline synthesis. I address the scheduling problem by *virtualizing* the memory interface to make memory accesses appear independent. Virtualization hides structural hazards and dependences between memory operations, allowing any conventional HLS tools to perform aggressive scheduling without the need for programmer intervention. Next, I introduce hazard resolution logic which resolves structural hazards via port arbitration and data hazards via pipeline squashing. The hazard resolution hardware is automatically generated based on the number of virtual memory ports, the

	Cycles																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
j=0	e.ld	v.ld	v.ld	v.st	v.st												
j=1					e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~								
j=2									e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~				
j=3													e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~

(a) Execution following static schedule in (b) incurs 17 cycles. `II=4` for all iterations.

	Cycles																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
j=0	e.ld	v.ld	v.ld	v.st	v.st												
j=1					e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~								
j=2							e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~						
j=3									e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~				

(b) Ideal execution incurs only 13 cycles. `II=2` after Cycle 4.

Figure 4.2: Execution of Maximal Matching — Assume a single-ported memory for each array. ~~`v.st`~~ indicates a `store` to array `v` that is not executed due to false conditional branch. (a) Execution following the static schedule in Figure 4.1(b). (b) Ideal latency-optimal execution.

type (read or write) of each port, and the possible data dependences between memory accesses.

While these proposed techniques are generally applicable to structural and data hazards for any expensive or limited hardware resources, this chapter emphasizes memory-related hazards, because memory ports constitute a scarce resource and memory dependences are a common limiting factor of pipeline throughput in irregular programs. In particular, this chapter focuses on irregular loops with conditional memory accesses and inter-iteration memory dependences whose access patterns cannot be asserted at compile time. The proposed approach works for truly dynamic data dependences for which speculation, hazard detection, and replay are necessary for high-throughput pipelined execution. These techniques provide the most performance benefit when the conditional accesses and data dependences are infrequent. The proposed approach is especially relevant as FPGA devices continue to attain higher memory bandwidths [DSB16]. Specifically, the major technical contributions of this chapter are threefold:

1. This chapter identifies a considerable performance gap in the HLS of irregular programs due to conservative nature of static pipelining in face of infrequent data-dependent dynamic hazards.
2. To my best knowledge, this is the first work to propose and study structural hazard resolution and speculative execution as dynamic pipelining techniques to bridge this performance gap.
3. This chapter describes techniques to compose generated RTL of hazard resolution logic with pipelines synthesized by a commercial HLS tool to achieve significant performance improvement on a suite of irregular benchmarks.

The remainder of the chapter is organized as follows: Section 4.1 illustrates the proposed dynamic hazard resolution techniques; Section 4.2 examines implementation details and discusses experimental results. Section 4.3 provides an overview of related work.

4.1 Dynamic Hazard Resolution for HLS

In this section, I propose three synergistic techniques to augment the HLS-synthesized pipeline with dynamic hazard resolution to address the performance gap caused by static alias analysis and scheduling. Figure 4.3 illustrates the overall architectural template for the augmented pipeline with Maximal Matching from Figure 4.1(a), composed of an accelerator synthesized with a virtualized memory interface connected to a hazard resolution unit (HRU) customized for the Maximal Matching application. The HRU can be further divided into a data hazard resolution unit (D-HRU) and a structural hazard resolution unit (S-HRU). The HRU dynamically resolves structural and data hazards that occur in the Maximal Matching pipeline and communicates with memory. The proposed approach does not require any modification to current pipelining algorithms. HRU logic is automatically generated based on the schedule of the synthesized pipeline and a set of may-alias memory access pairs obtained from static analysis and/or user-specified directives. The proposed techniques also benefit from more accurate alias analysis to decrease the number of may-alias pairs and reduce the complexity of the customized HRU. We will use Maximal Matching to illustrate the customizable architecture.

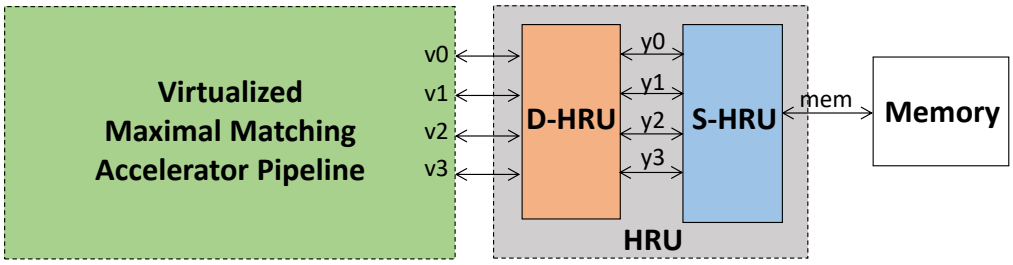


Figure 4.3: Architectural template for the composed Maximal Matching accelerator — HLS synthesized Maximal Matching pipeline with customized hazard resolution unit (HRU) consisting of a data hazard resolution unit (D-HRU) and a structural hazard resolution unit (S-HRU). A version with four virtual ports is shown.

4.1.1 Memory Interface Virtualization

Although we have identified a significant opportunity in improving the performance of synthesized pipelines by deferring the handling of infrequent hazards to runtime, we cannot take advantage of this opportunity unless we can easily reduce the pipeline `II` below what is deemed safe by the HLS tool. While it is possible to modify existing pipelining algorithms for this purpose, doing so would not be generally applicable to any HLS flows. It will also limit our ability to evaluate our techniques leveraging existing HLS tools.

I propose to relax infrequent resource and memory dependence constraints by *virtualizing* the memory interface to enable aggressive pipelining. Virtualization is a source-to-source transformation that alters each conditional or may-alias memory operations to access its own independent array. This technique decouples physical memory ports from the scheduling process to remove memory port constraints and inter-iteration memory dependences. In the perspective of the scheduler, the transformed memory operations do not share a common resource and thus do not alias. Hiding these infrequent hazards from the pipeline scheduler enables aggressive `II` reduction. Although the virtualized design contains more memory ports than the non-virtualized design, these ports interface with the HRU and will be arbitrated for actual physical memory ports, as shown in Figure 4.3.

To relax the constraints in Maximal Matching, I propose to virtualize its memory port interface by modifying the source code as shown in Figure 4.4 where the accesses to the same array `v` are transformed into accesses to four different arrays `v0`, `v1`, `v2` and `v3`. With this transformation, the HLS tool no longer sees the dependence between those memory operations and no longer encounters memory port conflict because each memory operation accesses a different array. Assuming two physical memory ports and the same schedule as that in Figure 4.1(b), Figure 4.5 shows the execution trace of the first few iterations for virtualized Maximal Matching pipelined to `II=2`. There exist two instances of potential dynamic data hazards between `v.st` in iteration `j=0` and `v.ld` in `j=1`.

```
for (j=0...num_edges){
 int s = e[j].src;
 int d = e[j].dst;

 if (v0[s]<0 && v1[d]<0){
  v2[s] = d;
  v3[d] = s;
}}
```

Figure 4.4: Maximal Matching example — Virtualized source code in C-like syntax. Compared to Figure 4.1(a), accesses to array `v` have been replaced by accesses to `v0`, `v1`, `v2`, and `v3`, respectively.

	Cycles									
	0	1	2	3	4	5	6	7	8	9
j=0	e.ld	v.ld	v.ld	v.st	v.st					
j=1			e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~			
j=2					e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~	
j=3							e.ld	v.ld	v.ld	...

Figure 4.5: Execution of virtualized Maximal Matching — With two physical memory ports and design pipelined to `II=2`. Note that there exist two instances of potential dynamic data hazard between `v.st` in `j=0` and `v.ld` in `j=1`.

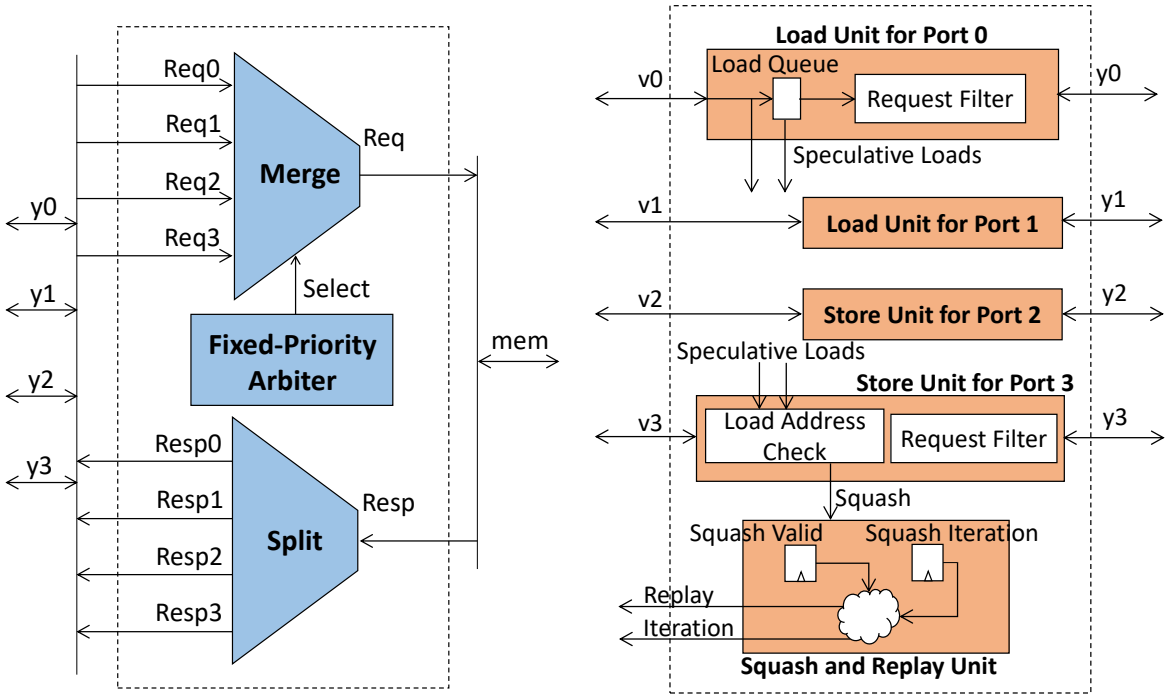
4.1.2 Structural Hazard Resolution

This section discusses the implications of aggressive scheduling on resources. Having bypassed unnecessarily conservative resource constraints during scheduling, it is necessary to complement the synthesized virtualized pipeline with an S-HRU to resolve structural hazards caused by infrequent conditional memory accesses that actually occur during runtime. While the proposed scheduling scheme with virtualization relaxes the constraints on memory ports, the number of physical memory ports is limited in reality. An S-HRU is required to appropriately arbitrate memory accesses that present at the virtual memory ports into a limited number of available physical memory ports. If there is only one physical memory port available for Maximal Matching, `v.st` from iteration `j=0` cannot execute in parallel with `v.ld` from iteration `j=1` as shown in Figure 4.6. In this case, the S-HRU prioritizes `v.st` in Cycle 3 and stalls `v.ld` until Cycle 4. Subsequent operations are similarly arbitrated and stalled. As shown in Figure 4.7(a), the S-

HRU implements a fixed-priority arbitration policy that always services request(s) from the earliest iteration(s). This policy preserves consistency for some speculatively executed memory accesses to reduce the need for squash-and-replay. The policy is also important for preventing deadlock and allowing the pipeline to flush in case of stall.

	Cycles									
	2	3	4	5	6	7	8	9	10	11
j=0	`v.ld`	`v.st`		`v.st`						
j=1	`e.ld`		`v.ld`		`v.ld`	~~`v.st`~~	~~`v.st`~~			
j=2					`e.ld`	`v.ld`	`v.ld`	~~`v.st`~~	~~`v.st`~~	
j=3							`e.ld`	`v.ld`	`v.ld`	...

Figure 4.6: Structural hazard resolution — With a single-ported memory, memory access from an earlier iteration is prioritized while others are stalled. `v.st` from `j=0` is prioritized over `v.ld` from `j=1` even though they are in the same cycle on the HLS-generated schedule. `v.ld` operations from `j=2` capture the unused bandwidth of `v.st` operations from `j=1` that are not executed.



(a) S-HRU (b) D-HRU

Figure 4.7: Hazard resolution units (HRUs) for Maximal Matching — (a) Structural hazard resolution unit (S-HRU). (b) Data hazard resolution unit employing speculative squash-and-replay (D-HRU).

While dynamic hazard resolution is able to arbitrate competing memory requests, it also allows an aggressively pipelined design to capture unused memory bandwidth when a conditional memory access is not executed due to a false conditional branch. As shown in Figure 4.6, dynamic memory port arbitration allows `v.ld` operations in iteration `j=2` to capture the unused memory bandwidth from `v.st` operations that are not executed in iteration `j=1` due to a false conditional branch. If the conditional accesses in Maximal Matching are infrequently executed, we can observe that the effective `II` will be very close to the target `II` of the aggressive pipeline. The application-specific S-HRU architecture automatically generated for Maximal Matching is shown in Figure 4.7(a) targeting one physical memory port. Because there are four independent array accesses in the virtualized design in Figure 4.4, the customized S-HRU is composed of a merge unit with four input buses (`Req0`, `Req1`, `Req2`, and `Req3`) that arbitrates four incoming memory requests from the virtual request ports of the accelerator to the single physical memory request port (`Req`). Similarly, the S-HRU also includes a split unit that routes any memory response from the single physical memory response port (`Resp`) back to the appropriate virtual response port (`Resp0`, `Resp1`, `Resp2`, or `Resp3`) of the accelerator. A fixed-priority arbiter determines the priority of requests in the same cycle by always servicing request from the earliest iteration.

The proposed approach is able to elastically adapt to memory bandwidth that varies over time. This is especially applicable to emerging accelerator-rich architectures where many accelerators share the same memory ports [CGG+14]. For these architectures, statically assigning memory ports is either inefficient or impractical. Section 4.2 shows that the proposed hazard resolution techniques can effectively adapt to varying memory bandwidth.

4.1.3 Data Hazard Resolution

In addition to resource constraints, the proposed aggressive scheduling scheme also relaxes inter-iteration dependence constraints by optimistically assuming that may-alias memory accesses would never alias. To ensure correct pipeline execution for the occasions when memory accesses do alias, however infrequent, the proposed approach further complements the synthesized virtualized pipeline with a D-HRU to resolve runtime

aliases not considered during static scheduling. In Figure 4.5, since the conditional accesses are actually executed in iteration `j=0`, `v.ld` in iteration `j=1` is executed at Cycle 3 before `v.st` in `j=0` is executed in Cycle 4. If the addresses of these may-alias memory accesses actually alias during runtime, the execution shown in Figure 4.5 would violate inter-iteration read-after-write dependence.

	Cycles									
	3	4	5	6	7	8	9	10	11	12
j=0	v.st	v.st								
j=1	~~v.ld~~	~~v.ld~~	e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~			
j=2		e.ld			e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~	
j=3							e.ld	v.ld	v.ld	...

Figure 4.8: Speculative squash-and-replay — The execution of `v.st` in Cycle 4 detects alias with `v.ld` executed in Cycle 3. Executed `v.ld` operations in Cycle 3 and 4 from `j=1` are squashed (indicated by ~~`v.ld`~~). Iterations `j=1` and onward are then replayed.

I propose to speculatively execute may-alias memory operations and perform squash and replay if the alias actually occurs during runtime. For Maximal Matching, we can speculatively execute the `load` operations from array `v` and squash and replay them only when memory aliasing is detected during the execution of a may-alias `store`. As shown in Figure 4.8, `v.ld` operations in iteration `j=1` execute speculatively but are later squashed when `v.st` in `j=0` executes in Cycle 4 and detects alias with the speculative `v.ld` from `j=1` executed in Cycle 3. Due to the squash, iteration `j=1` replays starting from Cycle 5, the cycle immediately after the alias is detected. On the other hand, `v.ld` operations executed speculatively in `j=2` do not get squashed because `v.st` operations in iteration `j=1` are not executed and cause no alias.

I propose a customized data hazard resolution unit with squash-and-replay capability (D-HRU) to enable a fully speculative pipeline. To prevent speculatively executed memory accesses from corrupting states, D-HRU is automatically generated to selectively include load queues and/or store queues to buffer speculatively executed requests until they are committed to memory. In addition, D-HRU selectively instantiates store-to-load forwarding unit to forward not yet committed store data. While the loads and stores reside in the queue, they are checked by other committing loads and stores to detect any

mis-speculation. D-HRU implements a squash-and-replay mechanism that is able to cancel and replay any mis-speculated iterations.

While the idea of speculation is borrowed from complex super-scalar processors, the customized architecture of an HLS synthesized pipeline provides a unique opportunity to greatly simplify the complexity of the speculation logic. As such, the hardware generation algorithm is designed to instantiate the minimum subset and minimum number of the aforementioned hardware modules to support the alias pattern of a particular application for a specific schedule. In Figure 4.7(b), the customized D-HRU instantiates a `Load Unit` for port `0` to buffer incoming load requests for array `v0` (`v[s].load`), because these requests may alias with store requests on port `3` from array `v3` (`v[d].store`). The size of the queue is determined by the difference between the worst-case schedule distance from the load to any potentially aliased stores (3 cycles in this case) and the `II` of the pipeline. Thus only one entry is needed in the load queue to buffer incoming requests at port `0` for `II=2`.

In Figure 4.7(b), the `Store Unit` for port `3` instantiates a `Load Address Check` unit that reads the speculative load addresses from the `Load Queue` for port `0` and check whether the current store request in port `3` (`v[d].store`) aliases with any speculative load requests from port `0` (`v[s].load`). If so, it sends a squash signal to the `Squash and Replay Unit` which squashes and replays the appropriate iterations. `Request Filter` is instantiated as part of the `Load Unit` or `Store Unit` to drop squashed requests. No store buffers need to be instantiated because the `store` operations are not speculatively executed. `Load Unit` for port `1` and `Store Unit` for port `2` implement load buffer and load check similar to those of port `0` and `3`, respectively.

4.2 Experiments

While virtualizing the memory interface is performed source-to-source, I develop a highly-parameterized hardware generation algorithm to automatically generate the minimum amount of HRU logic necessary for the particular application and compose the HRU logic with the synthesized pipeline to achieve high performance. The hardware generation algorithm leverages profiling and dependence analysis passes to extract infre-

quent may-alias memory access pairs, along with meta-data extracted from the schedule of the synthesized design to intelligently instantiate and connect HRU modules based on the architectural templates described in Section 4.1.

During hardware generation, the algorithm first extracts the necessary meta-data from the results of pipeline synthesis, including the `II` of the schedule, schedule distance between infrequent may-alias memory accesses, and the number of synthesized virtual memory ports. Then the algorithm generates the S-HRU based on the number of virtual and physical ports. Afterward, the algorithm instantiates a D-HRU if there exists dependence that must be resolved dynamically. The algorithm automatically customizes the composition of the D-HRU based on the number of virtual ports, a list of dependences, and the specification of each dependence. More specifically, a D-HRU is selectively composed of custom-size load/store queue, data forward unit, squash unit, replay unit, and filter units for resolving speculatively executed load and store operations.

The hardware generation algorithm is implemented within a Python-based hardware modeling framework, which supports concurrent structural hardware modeling and provides a collection of tools for simulating and translating Python RTL models to Verilog [LZB14]. To compose the synthesized pipeline with customized HRU, the algorithm instantiates a top-level model that integrates each HLS-generated accelerator design with appropriately-parameterized HRU models. Valid-ready interfaces are implemented to communicate between hardware units and stall the circuit when necessary. This composition can be simulated and synthesized with conventional tools. Both the C-level programs and composed RTL models are synthesized with Vivado 2015.1 targeting Xilinx Virtex-7 FPGA. QoR is obtained post place-and-route, and performance is obtained from cycle-accurate RTL simulation.

4.2.1 Benchmarks

To understand the implications of the proposed approach, we experiment with designs that exhibit data-dependent structural and data hazards from a range of application domains. The experiments emphasize irregularity typically not found in regular applications where current HLS tools excel. We discuss the following two applications in detail.

```
for (i=0; i<N; ++i){
 int m = feature[i];
 float wt = weight[i];
 if (m>THRESHOLD){
  float x = hist[m];
  hist[m] = x + wt;
}}
```

(a) `CountIf Histogram`

```
for (k=1; k<=m; k++){
 for (p=0; p<nz; p++){
  x[k][row[p]] +=
    a[p]*x[k-1][col[p]];
 }
}
```

(b) `Matrix Power`

Figure 4.9: Irregular loop kernels with conditional hazards — (a) `CountIf Histogram` constructs a weighted histogram of an array of features above a specified threshold. Array `hist` incurs conditional hazards. (b) `Matrix Power` computes the set of vectors $A^i\vec{x}$ for $i = [0, m]$. Array `x` incurs conditional hazards.

In Figure 4.9(a), each iteration of the `CountIf Histogram` kernel increases the bin indexed by the current feature value by adding the current weight if the feature value is above a specified threshold. There is an inter-iteration read-after-write dependence between the `load` on `line 5` and the `store` on `line 6`, which may cause data hazards if a subsequent iteration reads from the same histogram bin before the current iteration writes to it. While such memory aliasing is usually rare during histogram computation, it is impossible to assert the absence of such alias without prior knowledge of the sequence of feature values. Thus the HLS tool must create a conservative schedule such that a subsequent iteration reads from the histogram after the current iteration finishes writing to the histogram. Moreover, static scheduling unconditionally allocates a memory port for each memory access in a cycle even if the access is conditional. This results in inefficient utilization of memory bandwidth when the conditional accesses are predicated false at runtime.

In Figure 4.9(b), the `Matrix Power` kernel computes the set of vectors $A^i\vec{x}$ for $i = [1, m]$. With A stored as a coordinate list of `(row, column, value)` tuples, this kernel performs m sparse matrix-vector multiplications. The indirect memory accesses `x[k][row[p]]` and `x[k-1][col[p]]` on `line 3` and `line 4` present a potential inter-iteration read-after-write dependence because the result of $A^i\vec{x}$ depends on that of $A^{i-1}\vec{x}$. To ensure functional correctness without complete knowledge of the runtime values of `row[p]` and `col[p]`, the HLS tool must conservatively execute `load` from

`x[k-1][col[p]]` only after `store` to `x[k][row[p]]` from a previous iteration has been completed.

4.2.2 Results

In Table 4.1, we first compare the achieved latency, clock period, and resource usage between the baseline designs, alternative designs with S-HRU only, and alternative designs with both D-HRU and S-HRU. The baseline designs consist of the highest-throughput pipelines generated by Vivado HLS, while the alternative designs are virtualized versions of the baseline designs synthesized with the same commercial tool but augmented with dynamic hazard resolution. With a single-ported memory, Table 4.1 shows that the alternative designs are able to achieve a significant latency reduction compared to the baseline designs with reasonable timing and area overhead. Note that the Histogram design excludes the condition in Figure 4.9(a) to present a case in which S-HRU provides no benefit.

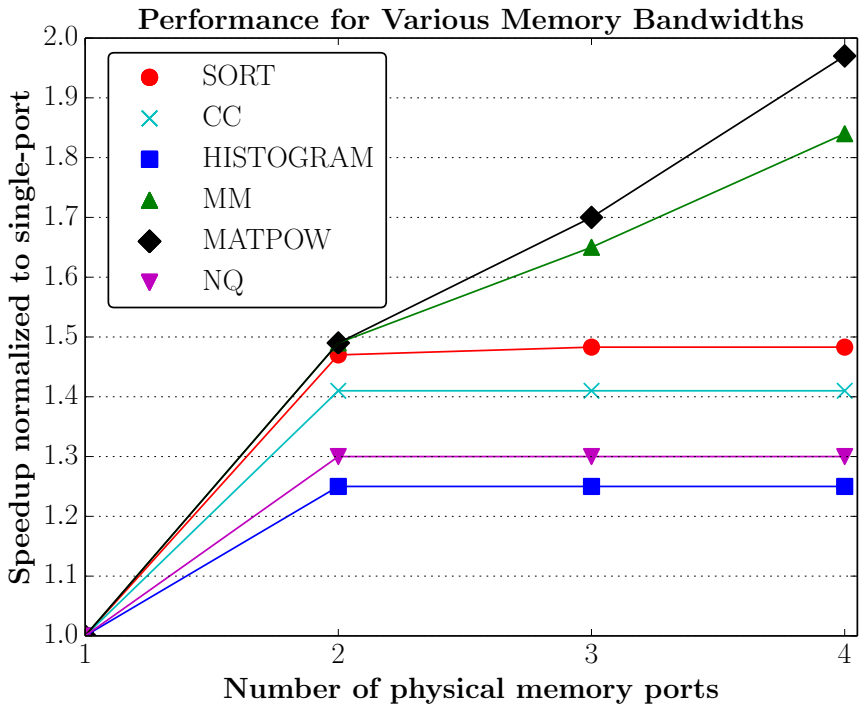


Figure 4.10: Performance comparison for different memory bandwidths — Speedup is normalized to the latency of single-ported memory case. The speedup saturates beyond two memory ports for designs with less pipelined parallelism or fewer memory accesses.

Table 4.1: QoR comparison between baseline (`base`), alternative with structural hazard resolution only (`s-hru`), and alternative with structural and data hazard resolution (`all`) using a single-ported memory. Target clock period is 5ns. Designs include Sorting (SORT), Connected Components Labeling (CC), Histogram (HIST), Maximal Matching (MM), Matrix Power (MATPOW), and N-Queens Algorithm (NQ).

Design	Latency (cycles)			Clock Period (ns)			#LUTs			#FFs			#DSPs		
	base	s-hru	all	base	s-hru	all	base	s-hru	all	base	s-hru	all	base	s-hru	all
SORT	3917	3153	3084	4.2	4.1	4.2	624	651	879	717	806	992	4	4	4
CC	2114	1513	1131	4.9	4.6	4.6	899	1047	1224	874	1308	1472	0	0	0
HIST	78014	78014	39382	3.9	4.1	4.2	508	592	652	529	602	753	2	2	2
MM	1973	1440	1150	3.9	4.3	4.6	487	586	960	479	516	852	0	0	0
MATPOW	16018	10529	6131	4.4	4.4	4.4	826	1789	1850	1228	2232	2413	13	12	12
NQ	2280	1344	1344	4.3	4.4	4.6	512	563	609	406	445	555	1	1	1

The amount of speedup is dependent on the input data pattern, number of executed conditional memory accesses, and the available physical memory bandwidth. Table 4.1 breaks down how latency improves with only structural hazard resolution and both structural and data hazard resolution. For Histogram, including only S-HRU provides no performance benefit because the pipeline throughput is limited by a long inter-iteration dependence cycle. In this case, it is necessary to incur the overhead of the D-HRU. On the other hand, N-Queens reaps no benefit from speculation because structural hazard resolution has indirectly helped resolve any dynamic data dependence due to the limited number of memory ports. In this case, it is sufficient to include only the S-HRU.

By studying different design points, Table 4.1 demonstrates the inherent trade-off between performance gain and area. techniques are important because loops that exhibit data-dependent hazards are often dominated by memory accesses. This is the reason that only a limited amount of compute resources are necessary. In addition, these loops usually contain only a couple of may-alias pairs, and thus require relatively lightweight hazard resolution logic that keeps timing and area well-contained.

We further study the effect of increasing memory bandwidth on performance and area by varying the number of physical memory ports. Figure 4.10 shows the speedup of each design for one to four memory ports normalized to the latency of the single-port case. For Sorting, Connected Components, Histogram, and N-Queens, performance saturates beyond two memory ports because these designs contain at most two unconditional memory accesses in each cycle most of the time. These two unconditional accesses need to be arbitrated only when there are less than two physical ports. Having more than two physical ports does not help because there aren't enough pipelined parallel accesses in these designs to utilize any ports beyond the two required. On the other hand, Maximal Matching and Matrix Power continue to reap the benefit of increasing memory bandwidth beyond two ports because both designs contain many memory accesses that can execute in parallel. Having a large number of memory accesses at each cycle allows the design to take full advantage of the available bandwidth.

Table 4.2 compares the achieved clock period and resource usage for different numbers of memory ports for Maximal Matching. Alternative designs with two to four ports incur `1.22x` to `1.70x` LUT counts and `1.02x` to `1.19x` FF counts with comparable tim-

ing. These overheads originate from the S-HRU shown in Figure 4.7(a) and apply equally to any benchmark. With an increasing number of physical memory ports, more complicated arbitration logic is needed to assign pending requests from the virtual ports to the available physical ports, which explains the increasing resource usage. According to Figure 4.10, Maximal Matching using four physical ports achieves over `1.8x` speedup compared to the single-ported case, which justifies the `1.70x` LUT and `1.19x` FF overhead.

Table 4.2: Timing and area overhead for increasing number of memory ports for Maximal Matching. No DSPs are used.

#Ports	Clock Period (ns)	#LUTs	#FFs
1	4.6	960	852
2	4.5	1171 (1.22x)	872 (1.02x)
3	4.5	1322 (1.38x)	941 (1.10x)
4	4.5	1629 (1.70x)	1010(1.19x)

4.3 Related Work

Many academic and commercial HLS tools, such as Vivado HLS [CLN+11] and LegUp [CCA+11], leverage static pipelining techniques to synthesize high-performance designs. Recent work in multithreaded pipelining [TLDZ14] extends these techniques to support dynamic memory behaviors. ElasticFlow enables the pipelining of irregular loop nests [TLZ+15]. Zhao et al. synthesize irregular program by decoupling data structures from algorithms [ZLS+16].

Alle et al. propose a runtime memory disambiguation technique where the address of a store is sent out before the store itself, allowing hardware to check whether an infrequently aliasing operation is expected to cause a hazard [AMD13]. This information is leveraged to enable more aggressive pipeline `II`. This chapter differentiates from this approach by considering structural in addition to data hazards for additional performance gain. This chapter also studies speculative execution to overcome the limitations pointed out by Alle et al.

Liu et al. extend polyhedral analysis to synthesizes pipelines that switches between aggressive (fast) execution, when hazards can be safely ignored, and conservative (slow) execution, when hazards are expected [LBC15, LWC16]. Unlike this class of non-speculative stalling approach, the proposed approach is this chapter does not require exact compile-time analysis to achieve high throughput. This chapter tackles dynamic hazard resolution more broadly by emphasizing sophisticated runtime mechanisms complemented by relatively simple compile-time analysis. Nevertheless, the approach proposed in this chapter can benefit from the compile-time analysis in Liu et al.

CHAPTER 5
FLUSHING-ENABLED LOOP PIPELINING

It is evident from Chapter 4 that HLS pipelining is conventionally not amenable to dynamic hardware behaviors because of its reliance on static scheduling techniques originating from the software pipelining domain [Rau94, SC95, CLS93]. While Chapter 4 shows that HLS pipelining lacks support for dynamic hazard resolution, this chapter demonstrates that HLS pipelining also lacks support for pipeline flushing due to the strict alignment among operations in a conventional HLS pipeline schedule. Because each operation must strictly be executed in its time slot designated by the modulo schedule, the entire pipeline must often be stalled in the presence of delay caused by unavailable data or variable-latency operations, severely hindering performance in many situations.

Stalling consists of freezing execution on the entire pipeline until all hazards have been resolved to prevent any unwanted behaviors. In the conventional loop pipelining context, stalling blocks execution for all in-flight iterations if any of those iterations experiences a delay. Therefore, previous in-flight iterations cannot finish unless the current iteration is no longer stalled. On the contrary, flushing-enabled loop pipelining allows unobstructed execution of previous iterations even when the current iteration is stalled. Resulting data get "flushed" out of the previous iterations even though subsequent iterations are essentially frozen. As an immediate benefit, flushing helps remove the unnecessary dependency among in-flight iterations caused by stalling.

This chapter studies the problem of flushing-enabled loop pipelining in HLS and explores the available options and limitations of different approaches. The major contributions are threefold:

1. To the best of my knowledge, this is the first work to (a) systematically study the problem of flushing-enabled loop pipelining in HLS and (b) propose three promising approaches to support pipeline flushing, including the dynamic approach, realigned approach, and unaligned approach.
2. This chapter present an analytical comparison on the performance of the proposed pipelining approaches, and discuss their advantages and disadvantages.

3. This chapter proposes an exact formulation and a novel heuristic algorithm for unaligned loop pipelining to minimize the potential resource conflicts due to flushing. Experimental results show that the proposed heuristic algorithm achieves near-optimal results.

The rest of the chapter will be organized as followed: Section 5.1 points out the difference between pipeline flushing and stalling; Section 5.2 describes the three proposed approaches to enable flushing in loop pipelining; Section 5.3 presents theoretical analysis of the proposed approaches; Section 5.4 reports experimental results; Section 5.5 discusses related work.

5.1 Preliminaries

Loop pipelining applies modulo scheduling [Rau94, Huf93, LGAV96] to construct a static schedule for a single loop iteration so that the same schedule can be repeated at a constant II, which dictates the upper bound on the pipeline rate and thus the overall throughput of the pipelined loop. If any iteration is delayed, such II would be violated, and the throughput would be negatively affected.

5.1.1 Definition of Throughput

Given a pipelined loop with an initiation interval of II, let $T(i)$ be the start time of the ith iteration, where $0 \leq i \leq N$.

Definition 1. *Throughput (TP) is the average number of iterations processed per clock cycle:*

$$TP = \frac{N}{T(N)} \tag{5.1}$$

Based on the definition above, throughput is decided by $T(N)$, the start time of the last iteration of the loop. In the case of normal loop execution without delay on any iteration, iteration i starts executing at time step $i \cdot II$, so $T(i) = i \cdot II$. Because $T(N) = N \cdot II$, the throughput is inversely proportional to II as expected. In the case of infinity loops ($N \to \infty$), the throughput becomes the reciprocal of the average latency between the start of successive iterations.

5.1.2 Pipeline Stalling

Throughput may be degraded when the pipeline is stalled. For example, if the input data interval is less than II, then it is not possible to periodically process a new iteration every II cycles. The attainable throughput would be determined by the rate of the slower-arriving data rather than the II.

There are many possible sources of pipeline stall in hardware synthesis. In general, pipeline stalls can be categorized under the following classes:

- *Input stalls*: Pipeline may be stalled when the input data rate is less than expected. In addition, the loop may contain variable-latency memory reads depending on the cache-memory hierarchy, so pipeline may also be stalled if such memory reads incur unexpected latency (e.g. cache miss).
- *Output stalls*: Pipeline may be stalled because it is attempting to write to a full FIFO or performing a variable-latency memory write.
- *Internal stalls*: Pipeline may be stalled because of data-dependent variable-latency operations, such as function calls and iterative divisions.

5.1.3 Pipeline Flushing

Although pipeline stall is usually enabled by default in conventional synthesis flow because it is least costly in area, it carries many undesirable and even unacceptable side effects that can be addressed by enabling flushing. For designs without continuously-running data, "flushing" out the end-of-stream under pipeline stall sometimes requires feeding additional "garbage" data depending on the depth of the pipeline. Otherwise, useful data may get stuck inside the pipeline. For a flushing-enabled pipeline, resulting data can continue to exit the pipe even if there is no new input. A prominent example involves video, where horizontal and vertical blanking introduce gaps in pixels [Fin10].

Among other pitfalls of pipeline stall, artificial deadlock is an important one in multi-block designs with insufficient buffers to balance the latency between different data flow paths. Figure 5.1 shows an example design with three pipelined blocks connected by FIFOs. The design contains a direct data path between block A and C as well as an indirect

data path that passes through block B. If the length of the FIFO on the direct path is insufficient in balancing the extra delay of the indirect path, block A's output data would be ready for block C through the direct path a number of clock cycles before block B's output data is ready for block C through the indirect path. Because both data inputs must be available for block C to execute, block C is stalled, and block A's data remain in the FIFO of the direct path. As block A is stalled because it is unable to output to a full FIFO, as shown in Figure 5.1, block A also stops outputting data to block B. Consequently, block B is also stalled, leaving data stuck in the pipeline. As a result, the design cannot be fully executed. Without the support for flushing, when one pipelined block is stalled due to insufficient buffering, the entire subsystem is deadlocked.

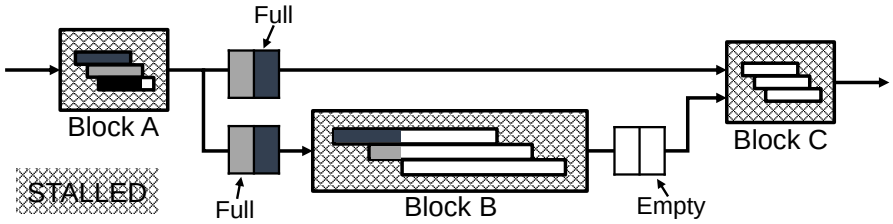


Figure 5.1: Multi-block design with deadlock

Issues seen with stalling render flushing not only desirable but necessary in implementing functionally correct, performance-driven designs. From resolving artificial deadlock to minimizing unnecessary inter-iteration dependency, enabling flushing in loop pipelining helps create a design that is more predictable in latency and more robust to potential hazards.

5.2 Flushing-Enabled Loop Pipelining

Enabling flushing in loop pipelining introduces the problem of resource collision because multiple operations may attempt to use the same resource at the same time. Figure 5.2 shows a conventional pipelined schedule for the loop body of a simple finite impulse response (FIR) filter with an II of 2. The DFG of the loop body is shown in Figure 5.2(a). As in Chapter 3, time steps within an II are called time slots. In Figure 5.2(b), LD_1 and LD_2 are scheduled in slot 1, and ST is scheduled in slot 2. Operations scheduled in the

same slot execute in parallel in the pipelined loop. Therefore, the availability of resources limits the operations that can be scheduled in the same slot. The schedule in Figure 5.2(b) is valid as long as operations always execute in their respective slots within the II-interval to maintain the required alignment as shown in Figure 5.3(a).

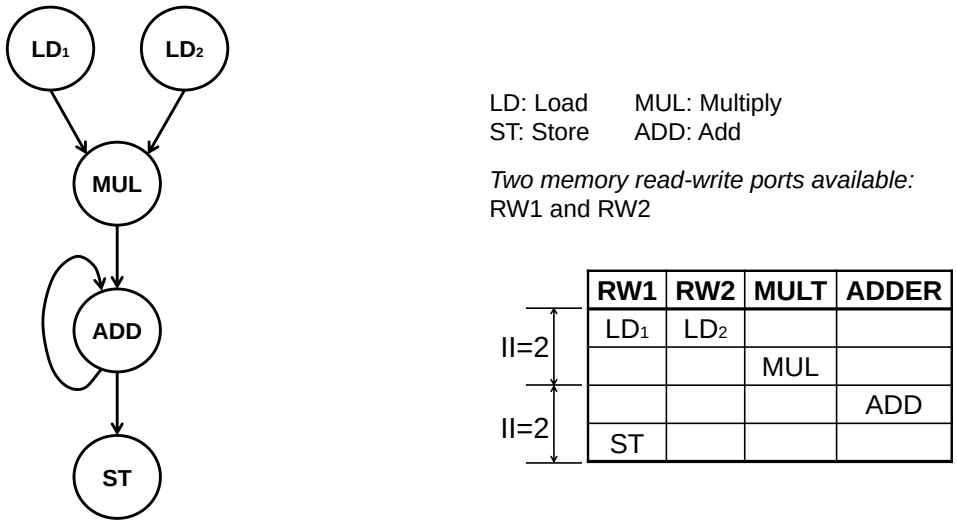
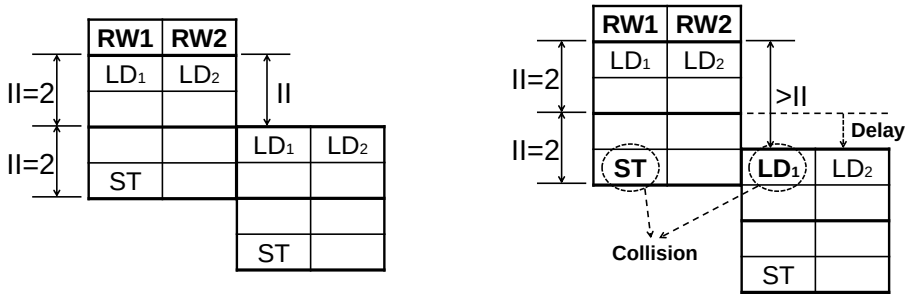


Figure 5.2: Loop body of simple FIR filter

As we can see, the alignment required by the pipelined schedule is violated in Figure 5.3(b), where a subsequent iteration is delayed due to delayed data availability. In a traditional pipeline without flushing, ST of the first iteration would have been stalled along with the second iteration. However, in a flushing-enabled pipeline, ST of the first iteration is executed in the original time step and flushed out the pipe even though the second iteration is delayed. In Figure 5.3(b), we see that flushing the first iteration leads to resource collision because there are three memory operations competing for only two memory ports at the fourth time step.

Therefore, the problem on hand is to enable flushing in loop pipelining while avoiding such resource collisions. Possible solutions include increasing II, dynamic realignment, dynamic collision resolution, and unaligned conflict-aware scheduling. By exploring both hardware and software-centric approaches, we offer solutions that encompass scheduling, binding, and RTL generation.



(a) (b)

Figure 5.3: Pipelined execution

5.2.1 Baseline Approach

One way to avoid resource collision is to reschedule the loop pipeline with a larger *II*, which decreases the degree of parallelism of the pipeline and effectively reduces the chance that parallel operations from different iterations compete for the same resource. For the example in Figure 5.2, increasing the *II* from 2 to 4, as in Figure 5.4(a), would completely resolve the resource collision presented in Figure 5.3(b). To determine the *II* that is collision-free for the current schedule, operations using the same resource must be scheduled within one *II*-interval. The worst case is to increase the *II* to the length of the loop body, which degenerates to a non-pipelined approach.

Simplicity is the key benefit of this approach. There is almost no need to modify the existing HLS infrastructure if we are just increasing *II*. However, following the baseline approach, *II* often needs to be increased significantly to completely avoid resource collision, resulting in detrimental degradation in throughput in many cases. It would be wise to find ways to enable flushing without having such a negative impact on *II*.

5.2.2 Realigned Approach

Instead of changing the *II* of the original pipelined schedule as for the baseline approach in Section 5.2.1, the realigned approach conserves the original schedule along with the original pipelined *II*. Instead of using a conservative measure and pessimistically increasing *II* to prevent resource collision, the realigned method takes a reactive approach by detecting collisions on-the-fly. To resolve these possible collisions, however, the re-

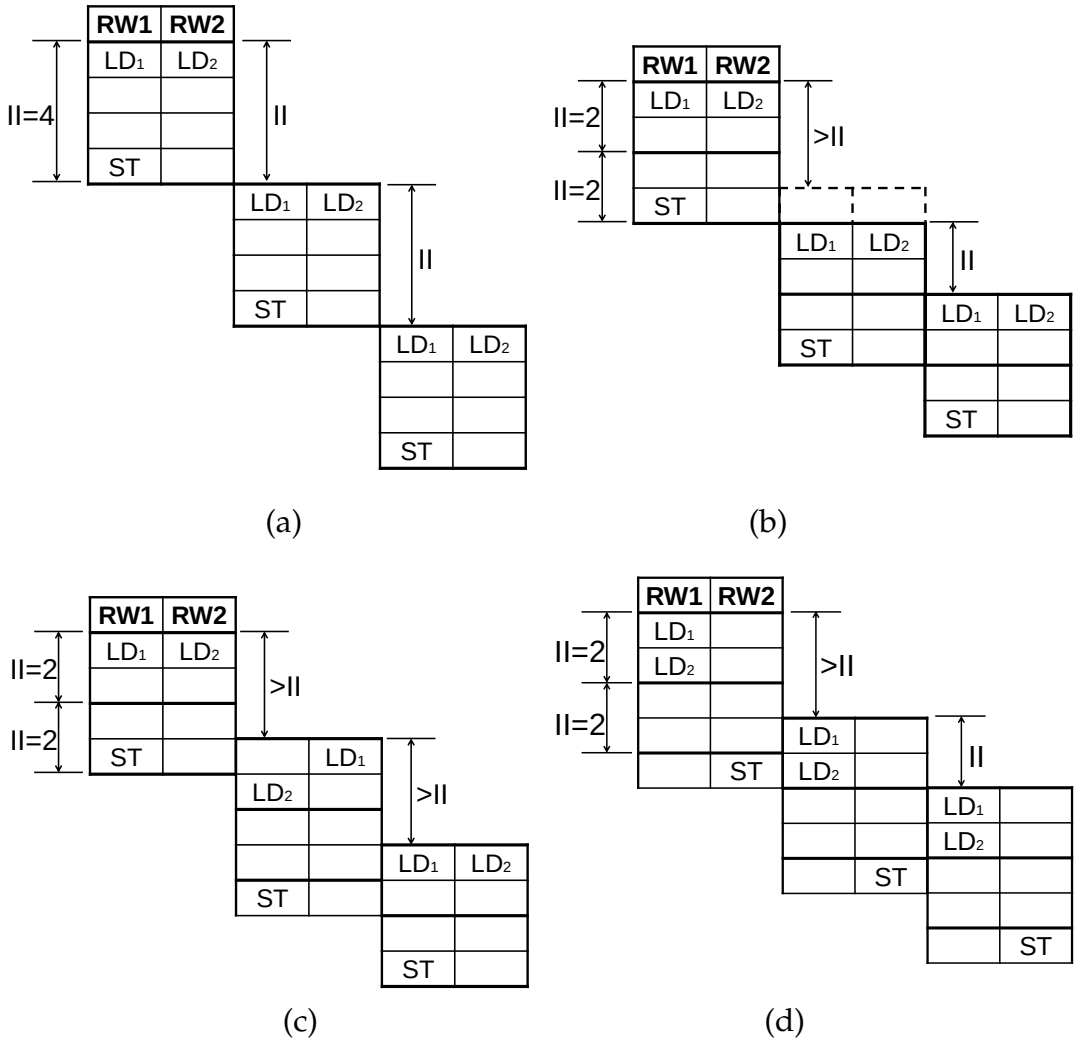


Figure 5.4: Flushing-enabled loop pipelining approaches

aligned approach adapts a lazy policy by simply waiting until the first available realigned slot based on the original pipelined schedule. As a result, the original slot assignments are always followed.

As shown in Figure 5.4(b), the collisions in Figure 5.3(b) is resolved by delaying the second iteration by one time step so the operations would execute in the correct time slot as specified by the original pipelined schedule in Figure 5.2(b). Unlike with simple pipeline stalling, ST of the first iteration is executed in its original time step and flushed out the pipe. As long as the delayed iterations are properly realigned, the pipeline would function correctly in the presence of hazards.

The realigned approach provides significant saving in the effective II compared to the baseline approach, although dynamic realignment might be costly in hardware. As discussed before, the saving in II translates to gain in throughput.

5.2.3 Dynamic Approach

Similar to the realigned approach in Section 5.2.2, the dynamic approach is reactive because it detects collisions on-the-fly. However, instead of waiting for the proper alignment, the dynamic approach eagerly resolves resource collision. Competing operations are arbitrated based on a priority that favors operations from earlier iterations, allowing earlier iterations to flush to its fullest extent in the presence of hazards.

For the resource collision scenario in Figure 5.3(b), ST and LD_1 are colliding in time step 4 and need to be arbitrated. Because the goal is to flush earlier iterations, ST would have priority over LD_1. The effective schedule after dynamic resource arbitration is shown in Figure 5.4(c). As we can see, we resolve collisions based on priority after collisions are detected, and operations are executed as early as possible, as soon as collisions are resolved.

As with the realigned approach in Section 5.2.2, the dynamic approach proposes modifications to the RTL generation of loop control logic as opposed to changing the scheduling algorithm. Therefore, it is not necessary to change the existing HLS scheduling technique for loop pipelining. Synthesized hardware for both the dynamic and realigned approaches must be able to detect resource collisions in real time. However, while we are adding realignment logic under the realigned approach, we are requiring arbitration logic under the dynamic approach. Compared to the baseline approach in Section 5.2.1, the dynamic approach also introduces significant saving in II and results in throughput gain. We will study throughput in more detail in Section 5.3.

5.2.4 Unaligned Approach

At the end, I propose a proactive approach for resolving potential resource collision while maintaining a desirable II for performance consideration. Termed the unaligned approach, this technique is proactive because it minimizes resource collision during HLS

scheduling instead of simply relying on hardware to detect collisions dynamically during execution as for the realigned and dynamic approach. For this purpose, I am proposing a new scheduling algorithm for flushing-enabled loop pipelining that is robust against potential resource collisions. Instead of increasing II and degrading performance, we can instead attempt to schedule around collisions.

Given: *(a)* A loop represented by a cyclic control data flow graph (CDFG) with dependences. *(b)* A target II. *(c)* A set of constraints including resource constraints, latency constraints, and relative timing constraints.

Goal: Generate a legal pipelined schedule with the target II while simultaneously determines binding to minimize the number of resources with potential resource collision.

The scheduling algorithm specifies the time step in which each operation should execute as well as the resource that the operation should use. For the example given previously (Figure 5.2), instead of scheduling the loop as in Figure 5.2(b) and resolving conflicts as in Figure 5.4(a), 5.4(b), and 5.4(c), the new algorithm would perform a collision-aware scheduling of the operations as in Figure 5.4(d), so that any delay in subsequent iterations would not cause resource collision with previous iterations. With the new scheduling algorithm, there is no longer any time step in our example that requires more than the number of available resources. As a result, there is no more resource collision.

Observation: Scheduling operations more than one II apart under the same resource leads to potential resource collisions.

The intuition behind the idea of one-II window can be derived from Figure 5.3(b). Without considering the delay of loop iterations, traditional modulo scheduling simply dictates that operations scheduled in the same time slot are not allowed to share the same resource. Now by also considering the possibility of delay of subsequent iterations, we must also make certain that operations scheduled more than one II apart in the pipelined schedule do not share the same resource. In Figure 5.3(b), because LD_1 and ST are both bound to the first read-write port and are scheduled more than one II apart, the delay of the second iteration causes a collision between LD_1 and ST at time step 4. If ST is instead scheduled within one II of LD_1, say in time step 2, then LD_1 of the second iteration cannot possibly collide with ST of the first iteration because the second iteration starts later than time step 2. Because iterations are spaced II time steps apart, and their delay would

only push them further down in time, resource collisions occur only between operations scheduled greater than or equal to one II apart.

To its best effort, the unaligned approach avoids such collisions by scheduling all operations under a particular resource within a one-II window as much as possible. In case a zero-collision schedule is not achievable, one can leverage the dynamic approach to resolve the unavoidable resource collisions.

Exact Formulation

I model the constraints set forth by this new scheduling algorithm with an ILP formulation in Equation (5.2). Let x_{ilk} be a binary variable that denotes whether operation i is scheduled at absolute time step l and executed by resource k. For a given II and available resources, this formulation computes a collision-aware schedule of at most L time steps that minimizes the number of resources with potential resource collision using the objective function in Equation (5.2a). c_k is a binary penalty variable used to indicate whether there is potential resource collision for resource k, and K represents the total number of available resources. Among other constraints related to dependency, timing, and binary assignment based on [ZL13] and [GAG94], Equation (5.2b) checks whether operations are scheduled within a one-II window on resource k, where t_f^k and t_l^k represent the time step in which resource k is first used and last used, respectively. A non-zero c_k indicates violation of the one-II window, meaning potential collision for resource k. Equation (5.2c) and (5.2d) denote the fact that any operations using resource k cannot be scheduled earlier

than t_f^k or later than t_l^k.

$$\text{minimize} \sum_{k=1}^{K} c_k \quad \text{subject to} \tag{5.2a}$$

$$t_l^k - t_f^k - L \cdot c_k < II \quad \forall k \tag{5.2b}$$

$$t_f^k - \sum_{l=1}^{L} l \cdot x_{ilk} - L\left(1 - \sum_{l=1}^{L} x_{ilk}\right) \leq 0 \quad \forall i,k \tag{5.2c}$$

$$\sum_{l=1}^{L} l \cdot x_{ilk} - t_l^k \leq 0 \quad \forall i,k \tag{5.2d}$$

$$\text{[Resource Constraints]} \tag{5.2e}$$

$$\text{[Dependency and Timing Constraints]} \tag{5.2f}$$

This optimization provides a scheduling and binding that minimizes the number of resources with potential resource collision.

Heuristic Algorithm

Because ILP is in general not scalable for large designs, I propose and implement a heuristic scheduling algorithm for the unaligned approach. Similar to the ILP formulation, the heuristic considers scheduling and binding simultaneously. Its algorithm prioritizes the scheduling of operations based on their heights and the concept of collision-aware mobility. Heights are calculated based on the height-based priority function used in [Rau94], which gives a good chance of scheduling operations in one pass. As mentioned previously, scheduling operations under the same resource within a one-*II* window insures that there will be no collision involving this resource. Therefore, to minimize the number of resources with potential resource collision while conserving resources, this algorithm attempts to schedule as many operations as possible within the one-*II* window of an already utilized resource before scheduling and occupying such window of a never utilized resource. Algorithm 2 outlines the scheduling heuristic for the unaligned approach.

When scheduling on an utilized resource, the algorithm tries to schedule the operation as close as possible to the earliest already scheduled operations on that resource, so operations are packed as closely and as tightly as possible into the same *II* window.

Algorithm 2 Scheduling heuristic for unaligned approach

while more operations need to be scheduled **do**
  Find operation i with maximum height and minimum mobility
  if i can be scheduled on an already utilized resource **then**
    Schedule i as close as possible to the earliest already scheduled operations on this existing resource
  else if i must be scheduled on a never utilized resource **then**
    Schedule i as close as possible to the centroid of subsequently scheduled operations on this new resource
  end if
end while

When scheduling on a never utilized resource, the algorithm would schedule the operation as close as possible to the centroid of the subsequently scheduled operations on that resource. The algorithm contains a mechanism in place to predict, based on the priority function, the operations that are most likely to be scheduled subsequently on a particular resource. Scheduling as close as possible to the centroid of subsequently scheduled operations again helps insure that operations under the same resource are as tightly packed as possible into the same II window. Without considering the centroid of subsequently scheduled operations, an operation may be scheduled on a resource at a time step far from the those of the norm, thereby immediately breaking the one-II window and rendering that resource useless for subsequent operations.

In this algorithm, collision-aware mobility is defined as the number of time steps for which an operation can be scheduled on a resource based on the current usage of the resource, the most updated earliest and latest possible scheduled times of the operations, and the one-II window based on the already scheduled operations. Formally, let L be the length of the loop in number of time steps, U_k be the set of time steps at which resource k is currently utilized, and $ASAP(i)$ and $ALAP(i)$ be the earliest and latest possible scheduled times, respectively, of operation i given the operations that are already scheduled. Then we define

$$\begin{aligned} M_i^k = \{x :\ & 1 \le x \le L, x \notin U_k, \\ & \max U_k - II < x < \min U_k + II, \\ & ASAP(i) \le x \le ALAP(i)\} \end{aligned} \tag{5.3}$$

where M_i^k is the set of time steps for which an operation i can be scheduled on resource k. Thus mobility m_i^k is defined as the size of the set: $m_i^k = |M_i^k|$.

5.3 Throughput Comparison of Proposed Approaches

In this section, we will analyze and compare the throughput of our proposed flushing-enabled loop pipelining approaches in the case of pipeline stalling. We mainly restrict our discussion to two representative scenarios of input delay. Nevertheless, this analysis can be generalized to other scenarios of delay. Results show that all of the proposed approaches can achieve high throughput in the case of slow-arriving input, while the unaligned approach may outperform the others in the case of variable-latency memory reads.

5.3.1 Throughput for Slow-Arriving Input

In order to achieve the best throughput, each iteration should start executing as soon as all its input data are available, and there is no resource collision. Let $E(i)$ be the earliest start time, the time at which input data become available to iteration i. $E(i)$ and the actual start time of iteration i, $T(i)$, should always satisfy the following conditions:

$$\begin{cases} T(i) \geq E(i), \\ T(i+1) - T(i) \geq II, \end{cases} \quad \forall i : 0 \leq i \leq N \tag{5.4}$$

To avoid potential resource collision, the realigned approach forces all iterations to start executing at aligned time steps, such that

$$T(i) \bmod II = 0, \forall i : 0 \leq i \leq N \tag{5.5}$$

Lemma 1. *For any iteration i, the latency between its actual start time $T(i)$ and its earliest start time $E(i)$ is less than II:*

$$T(i) - E(i) < II, \forall i : 0 \leq i \leq N \tag{5.6}$$

Equation 5.6 can be proved by induction based on Equations (5.4) and (5.5). The intuition behind this relation is that iterations can periodically catch up after the least common multiple of II and R cycles when the data interval R is larger than II.

Theorem 1. *In the case of slow-arriving inputs with a constant data interval, the realigned approach can achieve high throughput equal to the inverse of the data interval when the number of iterations is sufficiently large .*

Proof: Suppose that data inputs to successive iterations are arriving slowly with a data interval of R time steps per iteration for $R > II$. The earliest start time of the last iteration would be $E(N) = N \cdot R$, and the actual start time would be $T(N) < E(N) + II$.

$$TP^{R}_{\text{realigned}} = \frac{N}{T(N)} > \frac{N}{E(N) + II} = \frac{1}{R + \frac{II}{N}} \tag{5.7}$$

$$\lim_{N \to +\infty} TP^{R}_{\text{realigned}} = \frac{1}{R} \tag{5.8}$$

Since the data interval is R, there is no way to process more than one iteration every R time steps. Therefore, the attainable throughput would be $1/R$ under this scenario. As we can see, the realigned approach has achieved the attainable throughput.

Unlike the realigned approach, both dynamic and unaligned approaches try to start executing each iteration as soon as its dependent data is available. In the worst case, they have the same performance as that of the realigned approach. Because the realigned approach achieves the attainable throughput when N is sufficiently large, the other two approaches should also attain the same throughput when N is sufficiently large.

5.3.2 Throughput for Variable-Latency Memory Reads

When the pipeline is stalled because of variable-latency memory reads, all subsequent iterations would be delayed due to the pipeline stall. Assuming that the memory read can be either a cache hit or miss, let p ($p > 0$) be the cache miss penalty and r ($0 \leq r \leq 1$) be the miss rate, then the expected pipeline stall would be $r \cdot p$.

The realigned approach enforces that each iteration only start executing at aligned time steps to avoid resource collision. When there are p time steps of stalling because of cache miss in the current iteration, the next iteration may be delayed by more than p time steps to enforce the proper alignment with the II boundary. Suppose that the next iteration starts executing at the earliest subsequent aligned time step, the extra latency

incurred would be $L_{\text{realigned}} = \left\lceil \frac{p}{II} \right\rceil \cdot II$, and the expected throughput would be:

$$TP_{\text{realigned}} = \frac{r}{II + L_{\text{realigned}}} + \frac{1-r}{II},\ L_{\text{realigned}} = \left\lceil \frac{p}{II} \right\rceil \cdot II \tag{5.9}$$

The dynamic approach relies on the hardware to dynamically detect resource collision. When a memory read in the current iteration is stalled by p time steps, it always tries to start executing the next iteration as early as possible instead of waiting for the next realignment. In the best case, the next iteration would be stalled by exactly p time steps. However, if resource collision is detected at that time step, the iteration would be further stalled. In the worst case, the next iteration may be delayed until the earliest subsequent realigned slot, which results in the same scenario as the realigned approach. Based on this analysis, the extra latency incurred would be L_{dynamic} where $p \leq L_{\text{dynamic}} \leq L_{\text{realigned}}$, and the expected throughput would be:

$$TP_{\text{dynamic}} = \frac{r}{II + L_{\text{dynamic}}} + \frac{1-r}{II},\ p \leq L_{\text{dynamic}} \leq L_{\text{realigned}} \tag{5.10}$$

The unaligned approach relies on a static scheduling algorithm to minimize resource collisions when iterations start executing at unaligned time steps. If the scheduling algorithm can guarantee no resource collision, then the extra latency would be $L_{\text{unaligned}}$ where $L_{\text{unaligned}} = p$, and the expected throughput would be:

$$TP_{\text{unaligned}} = \frac{r}{II + L_{\text{unaligned}}} + \frac{1-r}{II},\ L_{\text{unaligned}} = p \tag{5.11}$$

Based on Equations (5.9), (5.10) and (5.11), the expected throughput depends on the miss rate and miss penalty. In case of low miss rate and high miss penalty, all three approaches yield similar throughput. On the contrary, if miss rate is high and miss penalty is low, then the unaligned approach outperforms the other two approaches. The above analysis can be generalized to pipeline stalls caused by other kinds of variable-latency operations.

5.4 Experimental Results

To demonstrate the practicability and scalability of our approach, we have prototyped the different loop pipelining techniques within a commercial HLS tool, and experimented

on real industry designs from multiple application domains, such as digital signal processing, image processing, and wireless communication. All designs shown in Table 5.1 and 5.2 target Xilinx Kintex 7 FPGA.

Table 5.1 compares the quality of results (QoR) between the realigned, dynamic, and unaligned approach, where each design is able to achieve the same pipelined *II* across all three approaches. The realigned approach usually has the least LUT and FF usage. The dynamic approach has the same schedule as the realigned approach; but the HLS tool needs to generate extra collision detection logic to avoid the collision at runtime. Such collision detection logic can be costly if there are many potential conflicts in the design. For example, the LUT count has increased about 34% in D1 and the timing has decreased about 10%. In general, the unaligned approach achieves a QoR between that of the realigned and dynamic approach. However, the unaligned approach may have a slightly longer latency when increasing latency is the only option to guarantee a collision-free design. Table 5.1 shows that the unaligned approach can get better timing than the dynamic approach, and can even be better than the realigned approach as for D3 and D5.

Table 5.2 shows the throughput comparison when there are input misses. Here the definition of throughput differs slightly from Equation 1. Instead of using iterations per cycle, we have also considered the frequency of the synthesized design and used iterations per second as the unit for throughput. As we have discussed in Section 5.3, the realigned approach has the worst throughput while the unaligned approach has the best one. For example, when p is 2 in D3, the throughput of the unaligned approach is 192% of the throughput of the realigned approach and 131% of that of the dynamic approach. Although the dynamic approach achieves higher number of iterations per cycle, the realigned approach is usually able to attain a better throughput because the realigned approach can achieve, in general, a higher clock frequency.

The algorithm for unaligned scheduling is further evaluated for different *II*s. Benchmark designs used include implementations of discrete cosine transform, a chemical plant controller, as well as digital signal processing algorithms, all commonly used to evaluate HLS tools. For ILP-based scheduling, the implementation generates the objective and constraints in a compatible format. Using the state-of-the-arts linear programming solver CPLEX [IBM17], we solve for an optimized schedule with the objective of minimizing

Table 5.1: QoR comparison between realigned, dynamic, and unaligned approaches

Design	*II*	#LUTs			#FFs			Latency (cycles)			Clock Period (ns)		
		Realigned	Dynamic	Unaligned	Realigned	Dynamic	Unaligned	Realigned	Dynamic	Unaligned	Realigned	Dynamic	Unaligned
D1	7	957	1284	1035	5270	5937	5905	102	102	105	4.1	4.5	4.4
D2	6	744	798	779	5898	6004	6301	25	25	28	6.9	7.1	7.1
D3	8	1266	1278	1305	1921	1981	1965	27	27	27	3.7	3.5	3.1
D4	4	948	1067	1038	1308	1343	1325	43	43	44	3.8	4.3	4.0
D5	4	589	635	648	786	826	986	22	22	23	3.5	3.8	3.0

Table 5.2: Throughput comparison between different approaches

Design	II	p	Throughput ($\times 10^6$ iterations / sec)		
			Realigned	Dynamic	Unaligned
D1	7	1	17.3	20.0	28.4
D2	6	1	12.0	17.6	20.1
D3	8	2	16.9	25.7	32.3
D4	4	1	32.9	29.1	50.0
D5	4	2	35.7	32.9	56.7

the number of resources with potential resource collisions. For heuristic scheduling, the proposed implementation calculates the heights and mobility of operations and follows Algorithm 2 to determine the schedule in one pass. Schedules are then validated, and resources with unresolved collisions are reported in Table 5.3.

The quality of the heuristic can be evaluated using Table 5.3 by comparing its results with the optimized results from the exact ILP formulation. In terms of the ability to resolve resource collisions, Table 5.3 shows that around 95% of the heuristic test cases report a number of resources with collision equal to or lower than that of the corresponding ILP test cases. The remaining report a slightly higher number of 1 resource with collision. In many cases for which the ILP solver times out not able to find a collision-free schedule, the heuristic performs better by providing a zero-collision schedule. In most cases, the heuristic finishes scheduling in a few seconds, while ILP runs for hours without finding a collision-minimized solution. The heuristic is able to mimic the optimized results of the exact formulation with a more reasonable runtime.

5.5 Related Work

Various forms of loop pipelining have been proposed for HLS in the past, such as loop winding [Gir87] and binding-aware pipelining [KLMW11]. Loop pipelining is also known as software pipelining [Lam88] in the compiler domain and has been widely used in modern compilers (e.g. GCC [HZ04]) to aggressively exploit instruction level parallelism across loop iterations. Modulo scheduling [RG81] is one of the most popular methods to enable software pipelining. Based on modulo scheduling techniques, sev-

Table 5.3: Comparing resource collisions between ILP and heuristic

Design	Number of Resources with Collisions							
	ILP				Heuristic			
	II=2	II=4	II=8	II=12	II=2	II=4	II=8	II=12
fir	0	1	0	0	0	1	0	0
arai	0	0	0	0	0	0	0	0
pr	0	0	0	0	0	1	0	0
wang	0	0	0	0	0	0	0	0
lee	0	0	0	0	0	0	0	0
mcm	0	0	0	0	0	0	0	0
honda	TO	0	0	0	0	0	0	0
chem	TO	TO	TO	TO	0	0	0	0

TO: ILP Solver Timeout

eral recent HLS systems have enabled loop pipelining to achieve better performance. For example, PICO-NPA [SAM+02] employs iterative modulo scheduling [Rau94] for synthesizing non-programmable loop accelerators; C-to-Verilog [BAMR10] performs modulo scheduling to reduce memory port usage under a fixed II constraint. Recently, Zhang and Liu [ZL13] extends SDC scheduling technique [CZ06] to minimize register pressure for loop pipelining; Morvan et al. [MDQ13] proposes a polyhedral-based pipelining technique for nested loops. The central idea of all these techniques is to periodically start executing a new iteration every II time steps. As a result, misalignment and flushing are not allowed. To my knowledge, this chapter presents the first systematic study of flushing-enabled loop pipelining in HLS.

CHAPTER 6
FAST AND ACCURATE ESTIMATION OF QUALITY OF RESULTS WITH MACHINE LEARNING

With good scheduling algorithms, HLS will be able to generate an optimized RTL design from a software program in respect to the constraints imposed during scheduling. The RTL then goes through logic synthesis, technology mapping, and PAR in a one-pass, non-iterative flow to generate the final layout for ASIC or bitstream for FPGA. These three steps are collectively known as implementation, as shown in Figure 6.1. On the surface, this flow provides a good recipe for achieving good QoR for the intended design given the well-tuned nature of today's EDA implementation tools.

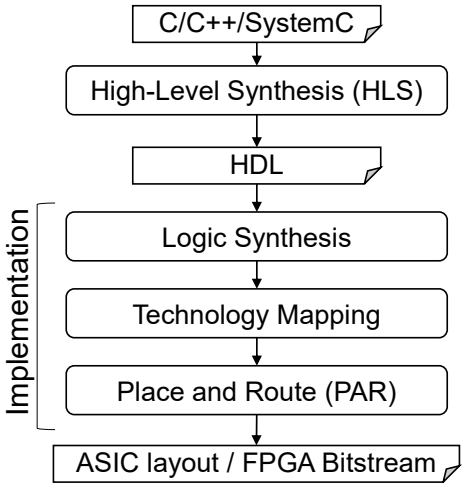


Figure 6.1: High-level design flow from software to hardware – HLS is followed by implementation to generate the final hardware.

Unfortunately, despite the effectiveness of implementation tools, HLS' reliance on implementation tools to synthesize the final design introduces an extra level of indirection, and thus inexactness, to the HLS optimization process. Although HLS is performed before implementation, HLS scheduling must optimize for post-implementation design quality to achieve the best design. To attain this goal, HLS needs a way to extract information on the effects of the implementation process on the design. One solution is for HLS to iteratively invoke the implementation flow to gather accurate information until the most optimal design has been reached. However, implementation typically takes hours if not days to complete, making this solution too time-prohibitive and counteractive to the pro-

ductivity benefit for which HLS is known. To avoid costly downstream implementation for every design choice, current HLS tools typically leverage pre-characterized QoR estimation models to approximate final design quality during the HLS scheduling process and base design decisions purely on these estimates. The adoption of pre-characterized QoR estimation models means that even the most exact scheduling algorithm will only be as good as the QoR approximations provided to it. The quality of the scheduled design depends heavily on the quality of QoR models in addition to the quality of the scheduling algorithm.

Despite the critical role of QoR estimation models in HLS, QoR models used in current HLS tools turn out to be highly inaccurate. Results from a collection of HLS benchmarks show that a commercial HLS tool overestimates the number of LUTs by an average of 4X and up to 40X. The same tool overestimates flip-flops by an average of 3X and up to 21X. The magnitude of the error indicates the lack of sufficiently accurate models for these estimation tasks, but is understandable because actual QoR depends on the cumulative effects of a number of complex transformations during implementation that are non-trivial to holistically characterize. Nevertheless, this presents a non-trivial gap between HLS-estimated QoR and actual post-implementation QoR. Due to this gap, HLS is relying on inaccurate QoR metrics to perform various optimizations whose perceived optimality may have little to do with the actual optimality of the design post-implementation. To deepen the inexactness, instead of constraining the scheduling process on actual target hardware resources (e.g., look-up table (LUT) and flip-flop (FF) for FPGA), HLS scheduling constrains on artificial RTL resources (e.g., adder, multiplier, memory port), which introduces an extra level of miscorrelation between HLS-perceived and actual post-implementation QoR. At the end, no matter how exact the scheduling algorithm becomes, inaccurate QoR estimation at the HLS stage remains the culprit compromising the grand promise of high-quality HLS-generated hardware.

To address this challenge, I propose to leverage 87 features that can be readily extracted from the HLS reports to accurately predict post-implementation results without actually running the implementation flow. Using these features, I propose to train a number of promising machine learning models, including linear regression, artificial neural network, and gradient boosted trees, to achieve high accuracy for the estimation tasks.

On top of accurate estimates, I carefully select models whose features can be directly interpreted to gain meaningful insights on the QoR estimation problem. This ensures that not only do we become equipped with accurate estimation models, but we also gain knowledge on the key contributing factors to QoR within the implementation process. I summarize the major contributions of this chapter as follows:

1. This chapter proposes to train a set of machine learning models that reduce the errors of HLS estimations by up to 138% using features extracted from HLS reports.
2. This chapter comparatively studies the trained models and employs domain-specific knowledge to explore model implications and predictive influence of various features.
3. I open-source this project, including the data as well as the pre-processing, training, testing, and analysis scripts, to enable further modeling and knowledge discovery efforts in the community.

Some existing work extracts the number of required functional units from DFGs and applies analytical models to approximate resource usage [SRWB14, MBNA17, ZFS+17]. While these techniques enable fast early-stage design studies prior to completing the HLS process, they are not designed to grasp the intricate effects of the implementation process. As a result, their estimations are only competitive against the crude HLS estimates this chapter aims to tackle. In contrast, this chapter proposes to leverage a data-driven approach to holistically model the combined effects of implementation to enable fast estimates that are competitive even against final implementation results.

The rest of this chapter is organized as follows: Section 6.1 provides further motivation for the problem; Section 6.2 describes our dataset and data processing; Section 6.3 illustrates machine learning models used to address the problem; Section 6.4 presents experimental results and insights; Section 6.5 reviews related work.

6.1 Motivation

As shown in Figure 6.2, an HLS-based design flow starts with a high-level software program, typically in C, C++, or SystemC, that is automatically synthesized into hard-

ware description language (HDL) models in Verilog or VHDL. HLS reports are generated alongside the models to indicate the expected performance, estimated resource usage and timing, as well as certain HDL details of the design. For an FPGA flow, the HDL models then run through logic synthesis, technology mapping, and place and route to generate a bitstream for the target FPGA. This HDL-to-bitstream process is collectively known as implementation as indicated in Figure 6.2. Implementation reports are generated to detail the actual resource usage and timing of the design on-chip.

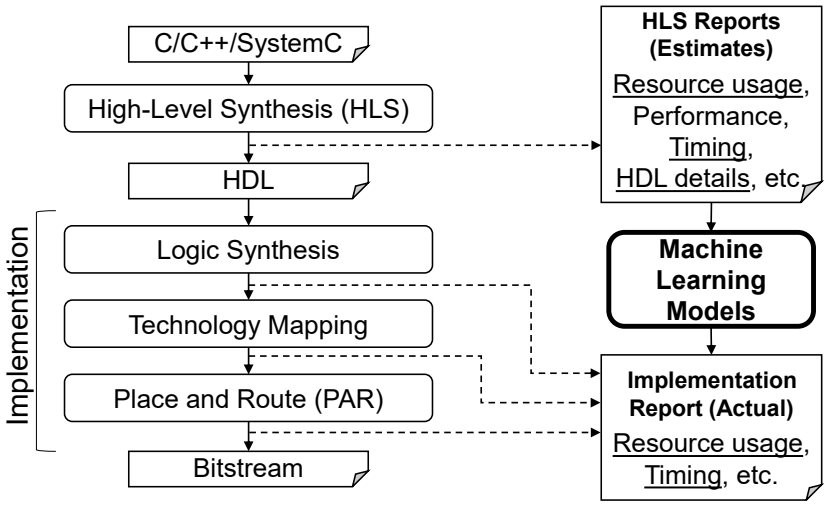


Figure 6.2: FPGA tool flow with HLS and proposed machine learning models (in bold) – The models use underlined metrics in the HLS reports to predict underlined post-implementation metrics.

Accurate estimations of post-implementation results at the HLS stage is difficult because the final implemented design reflects the cumulative effects of many non-trivial transformations through the series of implementation stages shown in Figure 6.2. Moreover, final resource usage and timing depend on constraints imposed by the target FPGA device, specifically the number, structure, and interconnection of device resources such as LUTs, FFs, DSP blocks (i.e., hardened multipliers), and block RAMs (BRAMs). To enable fast resource and timing estimation, HLS tools pre-characterize different functional units ahead of time and sum up the contributions of instantiated functional units during the synthesis of each design. However, such additive estimation approach fails to correctly capture the effects of post-HLS optimizations across functional units and neglects to consider limitations imposed by finite compute and routing resources on-chip.

As machine learning gains traction in design automation, I believe that it provides the means to holistically and precisely capture the multitude of factors affecting estimation accuracy. With the appropriate dataset on-hand, it is possible model the intricacies of the implementation process as a practical solution to the HLS estimation problem. As shown in Figure 6.2, this chapter proposes to apply machine learning models to predict actual resource usage from estimates that can be easily extracted at the HLS stage.

6.2 Data Processing and Analysis

To enable machine learning for HLS estimation, I build a dataset of HLS and implementation results consisting of over 1300 samples across 65 individual designs. To ensure quality, I leverage designs from well-known HLS benchmark suites, including CHStone [HTH+08], Machsuite [RAS+14], and S2CBench [SM14]. To increase diversity, I complement these benchmark suites with Rosetta benchmarks [ZGD+18], which include machine learning and real-time video processing applications. These additional designs differentiate from conventional benchmarks because they represent large fully developed applications instead of small kernel programs. They are implemented under realistic design constraints and reflect the latest application trends. Each design is run through the complete C-to-bitstream flow for various clock periods (1, 2, 3, 5, 10ns) targeting different FPGA devices (Xilinx Zynq, Artix7, Kintex7, and Virtex7). Table 6.1 summarizes the overall characteristics of the designs. The dataset can be further augmented by synthesizing the designs with different combinations of HLS optimization directives.

Table 6.1: Summary of designs – Post-implementation resource usages and worst negative slack (WNS) are shown. A negative WNS indicates that timing is not met.

	#LUT	#FF	#DSP	#BRAM	WNS (ns)
Max	63645	115452	795	350	8.4
Min	34	0	0	0	-39.7
Mean	5791	7395	25	19	-0.2

To construct the dataset, I first identify features of the HLS designs useful for predicting implementation results. For this purpose, I limit myself to features that can be readily

extracted from the HLS reports. As such, feature extraction incurs trivial computation overhead, and the proposed approach is generally applicable to different HLS tools. Similarly, I extract implementation results, known as the targets in this machine learning problem, from the implementation reports. After extraction, the dataset contains features and targets for each design sample and can be used to develop estimation models that map from features to targets.

6.2.1 Feature Extraction

While it is possible to leverage domain knowledge to hypothesize the relevance of different features to the estimation tasks, it is impossible to ascertain the predictive abilities of and relationship among the different features in advance. As a result, I extract as many relevant features as possible first and apply feature selection techniques (to be discussed in Sections 6.2.2 and 6.2.3) later to systematically remove any unimportant features. Feature extraction results in a total 234 features, all of which represent estimates from HLS reports.

6.2.2 Removing Redundant Features

This effort in building a comprehensive feature set may result in features that are statistically correlated and can be predicted with sufficient accuracy by other features. While this phenomenon of collinearity does not typically degrade the accuracy of estimation models, it nevertheless limits conclusions one can make about the predictive influence of a particular feature because the marginal contribution of the feature depends on which other correlated features are also present in the model. To overcome the effect of collinearity, I compute the Pearson's correlation coefficient for each pair of features on our dataset and select only one feature from each group of correlated features to be included for subsequent modeling.

6.2.3 Eliminating Irrelevant Features

Features that exert little influence on the targets should be eliminated to reduce the dimensionality of the data. Having fewer features leads to simpler models that require

shorter training time, reduce the chance of overfitting, and are easier to interpret. To eliminate irrelevant features, the data is fitted with a linear model with L1 regularization. As described later in Section 6.3.1, an L1-regularized linear model induces a sparse estimator that zeros out the coefficients of unimportant features and thus selects the important features (with non-zero coefficients). L1 feature selection is applied in conjunction with the correlation technique in Section 6.2.2 to reduce the number of features from 234 to 87. Table 6.2 describes categories of our selected features. For dimensionality reduction, L1 feature selection is chosen over matrix factorization approaches such as principal component analysis to preserve the original components of the feature set so that we can directly interpret the importance of each feature.

Table 6.2: Descriptions of categories of selected features

Category	Brief Description
Resource #	Usage & available number of each resource type.
Clock periods	Target clock period; achieved clock period & its uncertainty.
Logic ops	Bitwidth/resource statistics of logic operations (e.g., or, shift).
Arithmetic ops	Bitwidth/resource statistics of arithmetic operations (e.g., add, mul).
Memory	Number of memory words/banks/bits; resource usage for memory.
Multiplexer	Resource usage for multiplexers; multiplexer input size/bitwidth.

6.3 Estimation Models

Specifically, regression models are trained to estimate post-implementation resource usages for logic resources, including LUT and FF, because they exhibit significant inaccuracy. In general, regression is a supervised machine learning technique that infers a function from features to targets in the training set. For this study, there is a set of n training samples $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^{n}$, where $\mathbf{x}_i = [x_i^1, x_i^2, ..., x_i^p]^\top \in \mathbb{R}^p$ is the input vector of feature values for the ith sample, and $\mathbf{y}_i = [y_i^1, y_i^2, ..., y_i^q]^\top \in \mathbb{R}^q$ is the corresponding vector of target values. Here p denotes the number of input features (e.g., LUT count and clock period estimated by HLS), and q denotes the number of output targets (i.e., actual LUT and FF counts post-implementation). We further define $\mathbf{X} = [\mathbf{x_1}, ..., \mathbf{x_n}]^\top$ to denote feature values for all samples and $\mathbf{y^k} = [y_1^k, y_2^k, ..., y_n^k]^\top$ to denote values of target k for all samples. Each learning

task corresponds to one target estimation. A separate model f_k is trained for each target k, resulting in a set of mapping functions $\{f_k : \mathbb{R}^p \to \mathbb{R}\}_{k=1}^{q}$.

6.3.1 Linear Model

We start with the classic linear regression model, $\widehat{y_i} = \mathbf{x_i}^\top \mathbf{w}$, which models the target $\widehat{y_i}$ as a linear combination of features $\mathbf{x_i}$. Linear regression fits this model onto the training data to determine the $\mathbf{w}$ such that a loss function is minimized, where $\mathbf{w}$ represents the vector of coefficients for the learned model. In this study, we use the Lasso linear model with a loss function of $\left\|\mathbf{Xw} - \mathbf{y^k}\right\|_2^2 + \gamma\|\mathbf{w}\|_1$ to train a linear model that minimizes the least-square penalty on the training data with L1 regularization. By tuning the hyperparameter γ, the L1 regularization term $\gamma\|\mathbf{w}\|_1$ allows us to induce various degree of sparsity into $\mathbf{w}$ and in turn regulate the complexity of the model.

6.3.2 Neural Network

Unlike linear models, artificial neural network (ANN) is able to capture non-linearity in the data [HSW89]. An ANN consists of an input layer, followed by a series of hidden layers, and an output layer. Each hidden layer contains a set of neurons, each of which transforms values from the previous layer using a linear model followed by a non-linear activation function. While deep neural networks can represent complicated non-linear functions, a large amount of training data is needed for the model to converge. For the current estimation problem, the number of features is relatively small, and the amount of training data is limited. Therefore, ANNs with only a few fully-connected hidden layers are applied. Nevertheless, ANN is chosen to validate my hypothesis that the mapping from features to targets is non-linear. Compared to linear models, ANN requires tuning more hyperparameters (e.g., number of layers, neurons per layer) and results in non-convex loss functions that require more effort in training.

6.3.3 Gradient Boosted Trees

Based on building a "strong" regression tree by combining a series of "weak" ones, boosted trees model represents another promising non-linear technique [MBBF00]. It

models the target as the sum of regression trees, each of which maps the features to a score for the target. Target estimation is determined by accumulating scores across all trees. Gradient boosted trees implement gradient descent that optimizes the loss over the space of regression trees by repeatedly selecting the tree that points in the negative gradient direction. For this study, we apply Extreme Gradient Boosting (XGBoost) [CG16], a recent gradient tree boosting algorithm that enhances scalability using sparsity-aware approximate split finding. XGBoost has demonstrated accuracy competitive to neural networks while attaining better efficiency in both training and inference.

6.4 Experiments

Models described in Section 6.3 are implemented and trained in Python leveraging the scikit-learn [PVG+11] and XGBoost [CG16] libraries. All designs in the dataset are synthesized and implemented with Xilinx Vivado 2017.1 targeting Zynq-7000, Artix-7, Kintex-7, and Virtex-7. Experiments are performed on an Intel Xeon processor running at 2.5GHz. Regardless of the design, all models are able to complete the estimation tasks within milliseconds, compared to minutes or hours that each implementation stage typically incurs.

For regression, we compute the relative root square error (RRSE) defined as

$$\epsilon = \sqrt{\frac{||\hat{\mathbf{y}} - \mathbf{y}||_2}{||\mathbf{y} - \bar{\mathbf{y}}||_2}}$$

to evaluate and compare the accuracy of different models. RRSE is common metric for evaluating the quality of a regression model relative to that of a naive model that simply averages the target values. Here $\hat{\mathbf{y}}$ is a vector of values predicted by the model for a particular target, and $\mathbf{y}$ is a vector of actual ground truth values in the testing set for that target. $\bar{\mathbf{y}}$ denotes the mean value of $\mathbf{y}$. We randomly select 20% of our data as the testing set and perform random permutation cross-validation over 10 iterations on the remaining training/validation set. In each iteration, we randomly select 75% of the training/validation set for training and 25% for validation. While the validation set is used for parameter tuning to locate better models, the testing set remains isolated until the end only to evaluate the accuracy of the finalized models. This ensures that the models are

not tuned for the testing set. The training process employs grid search to find the best set of hyperparameters (e.g., γ in linear model, number of hidden layers for ANN) for each model.

6.4.1 Resource Estimation

Table 6.3 lists the estimation errors incurred by the HLS tool in comparison to those of our regression models. Based on this table, we observe that the HLS tool suffers from severe estimation errors for LUT and FF. Diving further into our dataset, we observe HLS estimated LUT counts that are on average 4.5x and up to 40x of actual LUT counts even for designs that utilize no BRAMs. This suggests that the disparity stems from the ineffectiveness of HLS additive estimation models (previously discussed in Section 6.1) in capturing the effects of logic synthesis and LUT/FF mapping and is not the result of failure in predicting whether BRAMs will be inferred.

Table 6.3: Resource estimation errors – RRSE incurred in the estimation of the usage of each resource type is shown. HLS Estimate: Models built-in to the HLS tool. Lasso, ANN, and XGB: Models described in Sections 6.3.1, 6.3.2 and 6.3.3, respectively. XGB+LASSO: Ensemble model consisting of XGB and LASSO.

Resource	LUT	FF
HLS Estimate	160%	136%
HLS+XGB+LASSO	28.1%	24.3%
HLS+XGB	29.9%	26.0%
HLS+ANN	44.5%	35.1%
HLS+LASSO	37.2%	34.7%

All the machine learning models studied are able to predict within milliseconds. Zooming in on a subset of our results, we see that while HLS suffers from severe error for estimating LUTs and FFs, all four of our models are able to reduce this error significantly to below 50%. While the linear models report an error of 37% for LUT and 35% for FF, the neural networks perform worse because they are more prone to overfitting. Instead, gradient tree boosting is the non-linear model that is able to improve upon linear model. The improvement of gradient boosted trees over linear model validates the hypothesis that the data are non-linear. Furthermore, if we add gradient tree boosting to the linear

model to capture non-linearity, we can reduce the error down to 28% for LUT and 24% for FF.

In addition, it would be a good idea to understand the spread of the model estimates in comparison to the ground truth values of the different targets. For this purpose, Table 6.4 reports the coefficients of determination for the different models to quantify the percentage of variance in the targets that are captured by the models [GS90]. A higher percentage in the table demonstrates that the actual post-implementation resource usage can be better replicated by the corresponding resource estimation model than a model with lower percentage. From the table, we see that the ensemble consisting of both gradient tree boosting and linear model is most superior in modeling the variance in the actual post-implementation resource usage.

Table 6.4: The percentage of variance in the targets that is captured by our models – Coefficients of determination are reported below to quantify how much of the variance in the target variable can be captured by our models using the features we selected. The models listed in the table are the same as those in Table 6.3.

Resource	LUT	FF
HLS+XGB+LASSO	92.1%	94.1%
HLS+XGB	91.1%	93.2%
HLS+ANN	80.2%	87.7%
HLS+LASSO	86.2%	88.0%

6.4.2 Model Interpretation

It is often desirable to interpret the trained models to understand key features that lead to good estimation. In fact, linear model and gradient boosted trees are included because both provide a weight for each feature indicating the feature's importance in the models. For example, it is possible to determine the importance of each feature in gradient tree boosting by the number of times that the feature is used as a split across all trees. Because of careful feature selection (Sections 6.2.2 and 6.2.3), we can interpret and discover knowledge from these models.

For clarity, Table 6.5 lists the most important categories of features for each of the estimation tasks in XGBoost. Not surprisingly, the post-implementation usage of each re-

source depends heavily on the corresponding built-in HLS estimates. This is expected because the proposed approach uses metrics in the HLS report as the starting point and is essentially "recalibrating" HLS-estimated metrics to match corresponding post-implementation QoR. Multiplexer-related features affect the number of LUTs and FFs because these resources are typically used for multiplexer implementation. Estimated clock period plays a role in estimating each resource because timing slack is a good indicator of how resources are mapped, placed, and routed. It is reasonable to see estimated clock periods as top features because they intuitively provide good indication of the difficulty of meeting timing. Features of logic operations are also important because they reflect the amount of inaccuracy introduced by the additive estimation model built-in to the HLS tool.

Table 6.5: Important categories of features for each estimation task in XGBoost – Ranked by combined importance of features in each category. #LUT, #FF, and #BRAM: HLS estimated resource counts. Mux: Multiplexer-related. Est_CP: Estimated clock periods. Logic_Ops: Logic operations.

Task	LUT	FF
Important Feature Categories	#FF	#FF
	#LUT	#LUT
	Mux	Mux
	Est_CP	Est_CP
	#BRAM	#BRAM

6.5 Related Work

Machine learning has been successfully applied within autotuning frameworks to effectively explore the large, high-dimensional space of tool-specific parameters controlling FPGA synthesis and implementation [YKNT16, XLZ+17]. For HLS, it has been leveraged for design space exploration to reduce the number of design candidates that need to run through the downstream implementation flow [LS16]. For resource estimation specifically, Koeplinger et al. learn three-layer ANN models to predict post-implementation resource usages from pre-characterized area models of a small set of architectural tem-

plates [KPZ+16]. Instead of template-based designs, techniques proposed in this chapter work for general HLS designs which are significantly more difficult to model. In addition, this chapter employs a larger and more complex set of designs to build the dataset and comparatively studies different regression models with more rigorous training, validation, and testing. This chapter also explores correlations and non-linearity within the data.

CHAPTER 7
CONCLUSION

Despite increasing adoption of HLS for its design productivity advantage, the lack of success in achieving high QoR out-of-the-box continues to hinder the productivity advantage for which HLS is known. While scheduling forms the algorithmic core that has greatly improved the quality of HLS-synthesized hardware in the past decade, the current scheduling methodology still presents a nontrivial set of QoR challenges that remain to be addressed. In this thesis, we term these challenges the QoR gap and attribute their causes to the *algorithm gap*, *flexibility gap*, and *estimation gap* inherent in today's HLS tools. We summarize the challenges as follows:

1. **Algorithm Gap from Inexact Static Scheduling** Current scheduling algorithms rely on fundamentally inexact heuristics and cannot accurately and globally optimize over a rich set of constraints. The lack of guarantee on optimality results in missing optimization opportunities and leaves open a QoR gap unknown in magnitude to both the designer as well as the HLS tool.

2. **Flexibility Gap from Lack of Support for Dynamic Scheduling** Current scheduling techniques rely on static compiler analysis and must make simplifying assumptions about statically unanalyzable program behaviors. Because of these assumptions, current HLS tools provide inadequate support for dynamic behaviors arising from variable-latency operations, irregular program patterns, and runtime data and control hazards.

3. **Estimation Gap from Disconnect between HLS and Downstream Tool Flow** Because HLS is unaware of the effects of downstream transformations on the HLS-synthesized design, QoR metrics relied upon by HLS turn out to be highly inaccurate when compared against actual post-implementation results. Without an accurate model for downstream QoR, designers and the HLS tool would be applying optimizations based on inaccurate information, resulting in designs with the wrong tradeoff.

7.1 Thesis Summary and Contributions

This thesis proposes coordinated static and dynamic scheduling to address the algorithm, flexibility, and estimation gaps inherent in current HLS tools. By leveraging new static scheduling algorithms, dynamic scheduling techniques, and machine learning to address each of the gaps, this thesis aims to close the overall QoR gap and achieve the overarching goal of attaining high QoR out-of-the-box for future HLS tools. This thesis begins with an introduction in Chapter 1 of today's key HLS challenges, including the algorithm, flexibility, and estimation gaps caused by inexact static scheduling, lack of support for dynamic scheduling, and disconnect between HLS and downstream implementation flow, respectively. These challenges help motivate the three important proposals of this thesis on closing these gaps, specifically guaranteeing the exactness of static scheduling with a joint SDC and SAT formulation, supporting dynamic scheduling with application-specific hazard resolution logic and intelligent scheduling, as well as leveraging a machine-learning-based approach to perform fast and accurate QoR estimation at the HLS stage.

For new static scheduling algorithms to address the algorithm gap, the thesis delves into the details of my proposed exact scheduling algorithm and framework with joint SDC and SAT, including detailed formulations for both the SDC and SAT portions. The thesis describes how to couple SDC and SAT in a loop so that the two formulations learn from each other in a conflict-driven manner to quickly prune the solution space and derive exact solutions. The thesis also describes various problem-specific techniques on accelerating the algorithm, including incremental SDC, incremental SAT, and specialized problem reduction. While Chapter 2 focuses on solving the unpipelined scheduling problem with this proposed algorithm and framework, Chapter 3 extends them to solving the pipelined scheduling problem.

In terms of dynamic scheduling to address the flexibility gap, Chapter 4 of the thesis describes a technique that combines scheduling and hardware generation to enable application-specific structural and data hazard resolution in an HLS-generated pipeline for irregular loops. Pipelines composed by the proposed technique has the ability to speculate, squash, and replay to achieve significant improvement in pipeline throughput.

Chapter 5 addresses the necessity of enabling flushing in HLS pipelines and studies three promising approaches for flushing-enabled pipelining in HLS. These approaches range from modifying the dynamic control logic of the pipeline to pure static scheduling of the program operations.

Finally, Chapter 6 of this thesis addresses the estimation gap by outlining how to leverage machine learning to bridge the disconnect between HLS and the downstream implementation flow. This chapter presents a set of machine learning models that are able to quickly predict post-implementation QoR using exclusively information available within the HLS stage. This technique can be leveraged by designer or the HLS tool during design optimization to more productively yet accurately zero in on the desirable design points without the pain of relying on the time-consuming implementation process.

The high-level contributions of this thesis can be summarized as follows:

1. **New Static Scheduling to Improve Algorithm** With a novel static scheduling algorithm, this thesis improves exact HLS scheduling by pushing the boundary of what is practically scalable and redefines the frontier between scalability and quality.
2. **Dynamic Scheduling for Greater Flexibility** This thesis enables dynamic HLS pipelining with the ability to handle runtime data and structural hazards in an application-specific and thus complexity-effective manner.
3. **Machine Learning to Enable Better Estimation** This thesis introduces machine learning as a promising alternative for EDA estimation tasks and demonstrates specifically its ability in improving otherwise inaccurate QoR estimates, a major pain point for HLS.

7.2 Thesis Impact

I believe that the technologies developed under this thesis will make their way into mainstream commercial EDA tools. In the short term, we are already seeing an influx of interests in applying machine learning to a variety of EDA problems, especially with the launch of DARPA's IDEA program to develop no-human-in-the-loop synthesis of source code to layout [Olo18]. As a major EDA vendor, Cadence has created a new research

and development program under IDEA to improve its tool, flow, and IPs with machine learning techniques [Cad19a]. Given my emphasis on training accurate and interpretable QoR models that are as simple as possible, there is very little barrier in quickly integrating these models into HLS tools. In fact, these models can be deployed into existing HLS tools by adding a simple post-synthesis step that extracts information from the conventional HLS report and uses the extracted information with the models to generate a more accurate report of QoR. However, since data for training these models reside mostly within the tool users instead of the EDA vendors, I believe that the users need to be proactive in collecting data and training these models. I imagine the scenario where each user company slowly builds up its own internal database of synthesis data and progressively refine its QoR models. Of course, this requires tool vendors to provide a convenience interface for accessing internal tool data and training various machine learning models.

Equivalently, it is inevitable that scheduling algorithms need to improve so that HLS can extend beyond regular dataflow-centric DSP applications to irregular applications such as graph algorithms, data analytics, and sparse tensor computations that are being emphasized as part of the big data movement. These applications require the flexibility provided by the proposed dynamic scheduling techniques. As more and more software and algorithm engineers adopt the HLS approach for lowering algorithms onto hardware as part of trend in software-defined hardware, we will see an increasing number of source descriptions that are not specifically tuned for hardware or not written with a well-defined hardware architecture in mind. This shifts the burden of designing an optimized architecture for possibly irregular programs onto the HLS tool, which in turn need to leverage dynamic scheduling to extract performance out of the irregularity. In terms of deploying to HLS tools, dynamic scheduling requires adding additional compiler analysis passes, complementing existing scheduling algorithms with proposed ones, and modifying the RTL generation process for supporting additional control logic in the FSMD.

Furthermore, the HLS market is no longer limited to small-scale FPGA-based IP vendors. Even major chip designer is expressing growing interest in integrating HLS into their chip design flow. Most prominently, NVIDIA has leveraged Catapult HLS to reduce design time from 14 to 9 months, simplify their code base by 5x, and speed up simulation by 1000x [Men19b]. Additionally, NVIDIA has released a set of open-source synthesizable

SystemC and C++ components for object-oriented hardware design with HLS [KKV+18]. As HLS becomes a core component of the mainstream design flow for major chip vendors, the tool must become more predictable in terms of QoR because design iterations are more expensive and respins are costly and highly undesirable. At the same time, HLS must ensure scalability and minimize tool runtime to meet internal schedules and time-to-market requirements. As a result, scalable exact scheduling serves as the cornerstone of mainstream HLS adoption. On top of that, accommodating more diverse use cases require that HLS consider additional constraints, such as those from power and physical design. Having SAT as part of the scheduling formulation, as in the proposed approach, provides the flexibility to model these constraints in the near future. In the context of current HLS tools, the proposed exact scheduling techniques can be deployed as additional scheduling engines complementary to the existing ones. They share the same inputs as the existing scheduling engines and produce the same format of output schedule for subsequent RTL generation. Therefore, users can elect to use these new schedulers whenever they need to satisfy more stringent QoR requirement or have a larger time budget to get better results.

7.3 Relevant Discussions

While most of the experimental results in this thesis focuses on targeting FPGA devices with FPGA based tool chain, the techniques proposed are not specific to FPGA and can be apply equally to ASIC based tools. State-of-the-art commercial and academic HLS tools perform scheduling targeting artificial intermediate resources (e.g., multipliers, memory ports) and rely on downstream logic synthesis to synthesize the generated RTL, composed of these intermediate resources, into actual target resources. For FPGA, the target resources include LUTs, FFs, DSPs, and BRAMs. For ASIC, the target resources include standard cells and SRAMs. Since HLS is only concerned about synthesizing the high-level program into an RTL consisting of intermediate resources, the HLS scheduling technique is agnostic to the final target technology. As a result, the proposed scheduling algorithms can be generally applied to both FPGA and ASIC designs. However, HLS requires timing and area estimates to optimize the generated RTL, as described in Chapter 6, and these

timing and area estimates are technology-dependent. The tool must characterize the timing and area of its set of intermediate resources for each target technology. Therefore, even though the scheduling algorithm is technology-agnostic, the input to the scheduling algorithm is technology-dependent, and the generated RTL is optimized only for the specific technology. This is where the proposed machine learning technique comes in to quickly and accurate predict timing and area metrics for each technology. Leveraging transfer learning technique, it is also possible to transfer knowledge from the model of one technology to that of another, so that we can derive the model of the new technology with only a few data points, resulting in significant saving in implementation runtime.

As it relates to HLS, there has always been a debate on what is the best language for design entry. Coming from a hardware background, an RTL design may prefer a bottom-up approach with SystemC, which provides C-based functionalities with common HDL features such as structural hierarchy, connectivity, and clock cycle accuracy. SystemC provides the option to select between untimed and timed description, implicit and explicit parallelism, as well a mix of the above depending on convenience for different parts of the design. Coming from a software background, a software engineer may prefer a top-down approach with C/C++, a completely untimed description with no explicit hierarchy or clocking. With the top-down approach, structural hierarchy, connectivity, and parallelism are partially specified with synthesis directives and partially inferred by the HLS tool. Regardless of which software language is used to describe the design, HLS scheduling is always needed to convert any untimed portion of the description into cycle-accurate RTL hardware. This scheduling process needs to be robust to correctly implement designer's intents, to achieve competitive QoR, and to not bottleneck other parts of the system.

Recently, there is a push toward even a higher level of abstraction with domain-specific languages (DSLs), which relies on compiling languages specialized for specific domain of applications to C/C++ that is then synthesized with conventional HLS tool. For example, there are active efforts in using Halide, an image processing DSL [RKBA+13], to implement optimized image processing pipelines on FPGA [PBY+17, Fix19]. Along with the great momentum behind deep learning, TVM and subsequently HeteroCL DSLs are also developed for specifying machine learning applications and optimizing them on FPGA [CMJ+18, LCH+19]. All these DSLs leverage a functional pro-

gramming framework to decouple computation from scheduling to enable various optimizations, such as loop unrolling and loop tiling, without sacrificing code clarity. HeteroCL additionally separates hardware customizations in compute, data type, and memory architecture to specifically provide a hardware-friendly programming environment. Regardless of how many layers of code transformations we add, the program is still lowered to some intermediate representation first and then synthesized by HLS. As such, HLS scheduling continues to have an important stake in determining the quality of the final design, no matter how the design is described and how optimized the design becomes in any higher-level languages.

7.4 Future Directions

Despite efforts described in this thesis, there remains a vast array of options in further improving scheduling to achieve top-notch QoR. As extensions to the proposed exact scheduling approach in Chapters 2 and 3, I believe that it is possible to further accelerate the performance of the solver by tightly coupling SAT and SDC within the conflict-driven search of the SAT solver itself. On a similar note, it is possible to accelerate ILP-based scheduling by injecting problem-specific knowledge into the branch-and-cut process commonly employed by ILP solvers [Mit02]. Similarly, it is a good idea to customize an SMT solver for the scheduling problem. The key to speedup stems from specializing the algorithm to the specific problem on hand. Because customization requires additional effort in comparison to directly making use of existing generic solvers, algorithmic specialization constitutes a tradeoff between performance and effort similar to that of hardware specialization.

While Chapter 6 has demonstrated the promise of machine learning in attaining accurate resource estimations at the HLS stage, it is just the very first step in providing accurate information for the HLS scheduling and synthesis process. Given that scheduling aims to achieve the best-performance design under constraints imposed by both resource and timing, additional effort must be allocated to address any lack of estimation accuracy for timing and performance. One immediate extension to techniques proposed in Chapter 6 is to incorporate structural information from CDFG to create more meaning-

ful features for the learning process. As opposed to the resource-oriented features used in Chapter 6, I hypothesize that structural information from CDFG is more indicative of timing-related characteristics of the design and would therefore be useful for discriminating designs with different timing and performance. In particular, I believe that graph embedding techniques [Hau99, LWH03, DDS16, NMV17, DZHL18] can be used to extract structural features from CDFG and map these features into low-dimensional feature matrices conducive to the proposed learning models. These features can also be combined with other metrics to further improve the accuracy of machine learning models.

Having demonstrated the importance of learning in exact scheduling methods for improving scalability and achieving good QoR in Chapters 2 and 3, it is also important to consider incorporating learning in heuristic scheduling algorithms. In the context of HLS scheduling, advancement in heuristics [CZ06, CBA14] have resulted in close to optimal QoR for many practical cases, with little cost on runtime. As such, learning can be used to close the final gap and achieve high-quality results without solving provably NP-hard problems. In fact, experts have pointed to many areas related to computer systems, including compilers and ASIC design, where machine learning can be used to make better decisions [Dea17]. Following this line of reasoning, I believe that we can combine what we learn from Chapters 2 and 3 with experiences from Chapter 6 to embed machine learning in scheduling heuristics to better guide the heuristic scheduling decisions. With accurate models for area, timing, and performance, machine learning can also be incorporated to optimize the design (possibly with SMT) over non-functional constraints on top of functional constraints to achieve better QoR [RGH+10]. Certainly, we can also embed machine learning in exact solvers (e.g., ILP, SAT) to improve pruning efficiency and convergence time.

BIBLIOGRAPHY

[AAHA+17a] T. Ajayi, K. Al-Hawaj, A. Amarnath, S. Dai, G. P. Davidson, S., G. Liu, A. Lotfi, J. Puscar, A. Rao, A. Rovinski, L. Salem, N. Sun, C. Torng, L. Vega, B. Veluri, X. Wang, S. Xie, C. Zhao, R. Zhao, C. Batten, R. Dreslinksi, I. Galton, R. Gupta, P. Mercier, M. Srivastava, M. Taylor, and Z. Zhang. Celerity: An Open-Source RISC-V Tiered Accelerator Fabric. *Hot Chips: A Symposium on High Performance Chips*, 2017.

[AAHA+17b] T. Ajayi, K. Al-Hawaj, A. Amarnath, S. Dai, S. Davidson, P. Gao, G. Liu, A. Rao, A. Rovinski, N. Sun, C. Torng, L. Vega, B. Veluri, S. Xie, C. Zhao, R. Zhao, C. Batten, R. G. Dreslinski, R. K. Gupta, M. B. Taylor, and Z. Zhang. Experiences Using the RISC-V Ecosystem to Design an Accelerator-Centric SoC in TSMC 16nm. *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[AG98] E. R. Altman and G. R. Gao. Optimal Modulo Scheduling through Enumeration. *Int'l Journal of Parallel Programming*, 1998.

[AMD13] M. Alle, A. Morvan, and S. Derrien. Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis. *Design Automation Conf. (DAC)*, 2013.

[BAMR10] Y. Ben-Asher, D. Meisler, and N. Rotem. Reducing Memory Constraints in Modulo Scheduling Synthesis for FPGAs. *ACM Trans. on Reconfigurable Technology and Systems*, 2010.

[Bie13] A. Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. *SAT Competition*, 2013.

[Cad19a] Cadence Design Systems. Cadence Selected for DARPA ERI Machine Learning Contract to Accelerate Electronic Design Innovation. `https://www.cadence.com`, Accessed: 2019.

[Cad19b] Cadence Design Systems. Status High-Level Synthesis. `https://www.cadence.com`, Accessed: 2019.

[CBA14] A. Canis, S. D. Brown, and J. H. Anderson. Modulo SDC Scheduling with Recurrence Minimization in High-Level Synthesis. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2014.

[CCA+11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2011.

[CD91] J. W. Chinneck and E. W. Dravnieks. Locating Minimal Infeasible Constraint Sets in Linear Programs. *ORSA Journal on Computing*, 1991.

[CG16] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. *Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, 2016.

[CGG+14] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman. Accelerator-Rich Architectures: Opportunities and Progresses. *Design Automation Conf. (DAC)*, Jun 2014.

[CHPM13] M. Chen, S. Huang, G. Pu, and P. Mishra. Branch-and-bound Style Resource Constrained Scheduling using Efficient Structure-Aware Pruning. *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2013.

[CLG02] J. Codina, J. Llosa, and A. González. A Comparative Study of Modulo Scheduling Techniques. *Int'l Conf. on Supercomputing*, pages 97–106, Jun 2002.

[CLN+11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2011.

[CLS93] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation Scheduling: A Loop Pipelining Algorithm. *Design Automation Conf. (DAC)*, Jun 1993.

[CMJ+18] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2018.

[CNK+12] T. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D. Singh, and S. Brown. OpenCL for FPGAs: Prototyping a Compiler. *Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2012.

[CZ06] J. Cong and Z. Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. *Design Automation Conf. (DAC)*, 2006.

[DDS16] H. Dai, B. Dai, and L. Song. Discriminative Embeddings of Latent Variable Models for Structured Data. *Int'l Conf. on Machine Learning*, 2016.

[De 94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[Dea17] J. Dean. Machine Learning for Systems and Systems for Machine Learning. *NIPS Workshop on ML Systems*, 2017.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 1962.

[DLZ18] S. Dai, G. Liu, and Z. Zhang. A Scalable Approach to Exact Resource-Constrained Scheduling Based on a Joint SDC and SAT Formulation. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[DLZZ17] S. Dai, G. Liu, R. Zhao, and Z. Zhang. Enabling Adaptive Loop Pipelining in High-Level Synthesis. *Asilomar Conf. on Signals, Systems, and Computers*, 2017.

[DMB11] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 2011.

[DSB16] M. Deo, J. Schulz, and L. Brown. Intel Stratix 10 MX Devices Solve the Memory Bandwidth Challenge. *Intel White Paper*, 2016.

[DTHZ14] S. Dai, M. Tan, K. Hao, and Z. Zhang. Flushing-Enabled Loop Pipelining for High-Level Synthesis. *Design Automation Conf. (DAC)*, 2014.

[DV15] C. Dasgupta and U. V. Vazirani. *Algorithms*, 2015.

[DXT+18] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, et al. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, 2018.

[DZ19] S. Dai and Z. Zhang. Improving Scalability of Exact Modulo Scheduling with Specialized Conflict-Driven Learning. to appear in *Design Automation Conf. (DAC)*, 2019.

[DZHL18] C. Donnat, M. Zitnik, D. Hallac, and J. Leskovec. Learning Structural Node Embeddings via Diffusion Wavelets, 2018.

[DZL+17] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang. Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[DZZ+18] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Young, and Z. Zhang. Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2018.

[ED97] A. E. Eichenberger and E. S. Davidson. Efficient Formulation for Optimal Modulo Schedulers. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 1997.

[EDA14] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Optimum Modulo Schedules for Minimum Register Requirements. *Int'l Conf. on Supercomputing*, 2014.

[Fin10] M. Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corp., 2010.

[Fix19] Fixstarts Solutions. Halide to FPGA Beta. `https://www.halide2fpga.com`, Accessed: 2019.

[For05] J. Forrest. CBC User Guide. *IBM Research*, 2005.

[GAG94] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining. *Int'l Symp. Microarchitecture*, 1994.

[GDZ15] U. Gupta, S. Dai, and Z. Zhang. Rosetta: A Realistic Benchmark Suite for Software Programmable FPGAs. *Suite of Embedded Applications and Kernels Workshop (SEAK)*, 2015.

[Gir87] E. Girczyc. Loop Winding – A Data Flow Approach to Functional Pipelining. *Int'l Symp. Circuits and Systems*, May 1987.

[GS90] S. A. Glantz and B. K. Slinker. *Primer of Applied Regression and Analysis of Variance*. McGraw-Hill Higher Education Medical, 1990.

[Hau99] D. Haussler. Convolution Kernels on Discrete Structures. Technical report, Department of Computer Science, University of California at Santa Cruz, 1999.

[Hor10] A. Horbach. A Boolean Satisfiability Approach to the Resource-Constrained Project Scheduling Problem. *Annals of Operations Research*, 2010.

[HSW89] K. Hornik, M. Stinchcombe, and H. White. Multilayer Feedforward Networks are Universal Approximators. *Neural networks*, 1989.

[HTH+08] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. CHStone: A Benchmark Program Suite for Practical C-Based High-Level Synthesis. *Int'l Symp. on Circuits and Systems (ISCAS)*, 2008.

[Huf93] R. A. Huff. Lifetime-Sensitive Modulo Scheduling. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 1993.

[HZ04] M. Hagog and A. Zaks. Swing Modulo Scheduling for GCC. *GCC Developers' Summit*, Jun 2004.

[IBM17] IBM ILOG CPLEX. IBM ILOG CPLEX Optimization Studio CPLEX User's Manual Version 12 Release 8. *International Business Machines Corporation*, 2017.

[Int18] Intel HLS. UG-20037: Intel High Level Synthesis Compiler User Guide. *Intel*, 2018.

[JBI17] L. Josipovic, P. Brisk, and P. Ienne. From C to Elastic Circuits. *Asilomar Conf. on Signals, Systems, and Computers*, 2017.

[JDSZ18] Z. Jiang, S. Dai, G. E. Suh, and Z. Zhang. High-Level Synthesis with Timing-Sensitive Information Flow Enforcement. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[JZPC08] W. Jiang, Z. Zhang, M. Potkonjak, and J. Cong. Scheduling with Integer Time Budgeting for Low-Power Optimization. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2008.

[KKV+18] B. Khailany, E. Krimer, R. Venkatesan, J. Clemons, J. S. Emer, M. Fojtik, A. Klinefelter, M. Pellauer, N. Pinckney, Y. S. Shao, S. Srinath, C. Torng, S. L. Xi, Y. Zhang, and B. Zimmer. A Modular Digital VLSI Flow for High-Productivity SoC Design. *Design Automation Conf. (DAC)*, 2018.

[KLMW11] A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe. Realistic Performance-Constrained Pipelining in High-level Synthesis. *Design, Automation, and Test in Europe (DATE)*, Mar 2011.

[KPZ+16] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. *Int'l Symp. on Computer Architecture (ISCA)*, 2016.

[Lam88] M. S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 1988.

[LBC15] J. Liu, S. Bayliss, and G. A. Constantinides. Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2015.

[LCH+19] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[LGAV96] J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. *Conf. on Parallel Architectures and Compilation Technique*, 1996.

[LS16] D. Liu and B. C. Schafer. Efficient and Reliable High-Level Synthesis Design Space Explorer for FPGAs. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2016.

[LSVH18] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen. Are We There Yet? A Study on the State of High-Level Synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.

[LTD+17] G. Liu, M. Tan, S. Dai, R. Zhao, and Z. Zhang. Architecture and Synthesis for Area-Efficient Pipelining of Irregular Loop Nests. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2017.

[LVAG95] J. Llosa, M. Valero, E. Ayguadé, and A. González. Hypernode Reduction Modulo Scheduling. *Int'l Symp. on Microarchitecture (MICRO)*, 1995.

[LWC16] J. Liu, J. Wickerson, and G. Constantinides. Loop Splitting for Efficient Pipelining in High-Level Synthesis. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, May 2016.

[LWH03] B. Luo, R. C. Wilson, and E. R. Hancock. Spectral Embedding of Graphs. *Pattern recognition*, 2003.

[LZB14] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[MBBF00] L. Mason, J. Baxter, P. L. Bartlett, and M. R. Frean. Boosting Algorithms as Gradient Descent. *Advances in Neural Information Processing Systems (NIPS)*, 2000.

[MBNA17] M. Makni, M. Baklouti, S. Niar, and M. Abid. Hardware Resource Estimation for Heterogeneous FPGA-based SoCs. *Symp. on Applied Computing (SAC)*, 2017.

[MDQ13] A. Morvan, S. Derrien, and P. Quinton. Polyhedral Bubble Insertion: A Method to Improve Nested Loop Pipelining for High-Level Synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 32(3):339–352, 2013.

[Men19a] Mentor Graphics. Catapult High-Level Synthesis. `https://www.mentor.com`, Accessed: 2019.

[Men19b] Mentor Graphics. High-Level Synthesis and Lower Power: NVIDIA - HLS. `https://www.mentor.com`, Accessed: 2019.

[Mit02] J. E. Mitchell. Branch-and-cut Algorithms for Combinatorial Optimization Problems. *Handbook of Applied Optimization*, 2002.

[MZ09] S. Malik and L. Zhang. Boolean Satisfiability from Theoretical Hardness to Practical Success. *Communications of the ACM*, 2009.

[Nie12] R. Nieuwenhuis. SAT and SMT are Still Resolution: Questions and Challenges. *Automated Reasoning*, 2012.

[NMV17] G. Nikolentzos, P. Meladianos, and M. Vazirgiannis. Matching Node Embeddings for Graph Similarity. *Conf. on Artificial Intelligence (AAAI)*, 2017.

[NR01] M. Narasimhan and J. Ramanujam. A Fast Approach to Computing Exact Solutions to the Resource-Constrained Scheduling Problem. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 2001.

[OKROS16] J. Oppermann, A. Koch, M. Reuter-Oppermann, and O. Sinnen. ILP-based Modulo Scheduling for High-Level Synthesis. *Intl'l Conf. on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, 2016.

[Olo18] A. Olofsson. Intelligent Design of Electronic Assets (IDEA) & Posh Open Source Hardware (POSH). *DARPA/MTO*, 2018.

[PBY+17] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. on Architecture and Code Optimization (TACO)*, 2017.

[PPM86] A. C. Parker, J. T. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. *Design Automation Conf. (DAC)*, 1986.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 2011.

[PZSC13] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-Based Data Reuse Optimization for Configurable Computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2013.

[RAS+14] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. Machsuite: Benchmarks for Accelerator Design and Customized Architectures. *Int'l Symp. on Workload Characterization (IISWC)*, 2014.

[Rau94] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. *Int'l Symp. on Microarchitecture*, Nov 1994.

[RG81] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. *ACM SIGMICRO Newsletter*, 1981.

[RGH+10] F. Reimann, M. Glaß, C. Haubelt, M. Eberl, and J. Teich. Improving Platform-Based System Synthesis by Satisfiability Modulo Theories Solving. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010.

[RJ94] M. Rim and R. Jain. Lower-bound Performance Estimation for the High-Level Synthesis Scheduling Problem. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 1994.

[RKBA+13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPLAN Notices*, 2013.

[RSJM99] G. Ramalingam, J. Song, L. Joskowicz, and R. E. Miller. Solving Systems of Difference Constraints Incrementally. *Algorithmica*, 1999.

[Sal16] L. Salmon. Circuit Realization at Faster Timescale (CRAFT). *DARPA/MTO*, 2016.

[SALL15] S. Skalicky, T. Ananthanarayana, S. Lopez, and M. Lukowiak. Designing Customized ISA Processors using High Level Synthesis. *Int'l Conf. on Reconfigruable Computing and FPGAs (ReConFig)*, 2015.

[SAM+02] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing*, 2002.

[SC95] F. Sánchez and J. Cortadella. Time-Constrained Loop Pipelining. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 1995.

[SDMZ17] N. K. Srivastava, S. Dai, R. Manohar, and Z. Zhang. Accelerating Face Detection on Programmable SoC Using C-Based Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[SM14] B. C. Schafer and A. Mahapatra. S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis. *IEEE Embedded Systems Letters (ESL)*, 2014.

[SRWB14] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. *Int'l Symp. on Computer Architecture (ISCA)*, 2014.

[Tar72] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1972.

[TDGZ15] M. Tan, S. Dai, U. Gupta, and Z. Zhang. Mapping-Aware Constrained Scheduling for LUT-Based FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2015.

[TLDZ14] M. Tan, B. Liu, S. Dai, and Z. Zhang. Multithreaded Pipeline Synthesis for Data-Parallel Kernels. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2014.

[TLZ+15] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang. ElasticFlow: A Complexity-Effective Approach for Pipelining Irregular Loop Nests. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2015.

[VL81] J. Van Loon. Irreducibly Inconsistent Systems of Linear Inequalities. *European Journal of Operational Research*, 1981.

[WIGG05] C. Wang, F. Ivančić, M. Ganai, and A. Gupta. Deciding Separation Logic Formulae by SAT and Incremental Negative Cycle Elimination. *Logic for Programming, Artificial Intelligence, and Reasoning*, 2005.

[XLZ+17] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang. A Parallel Bandit-Based Approach for Autotuning FPGA Compilation. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[YKNT16] Q. Yanghua, N. Kapre, H. Ng, and K. Teo. Improving Classification Accuracy of a Machine Learning Approach for FPGA Timing Closure. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2016.

[ZCDC15] Z. Zhang, D. Chen, S. Dai, and K. Campbell. High-Level Synthesis for Low-Power Design. *IPSJ Transactions on System LSI Design Methodology (T-SLDM)*, 2015.

[ZFS+17] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He. COMBA: A Comprehensive Model-based Analysis Framework for High Level Synthesis of Real Applications. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2017.

[ZGD+18] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. Velasquez, W. Wang, and Z. Zhang. Rosetta: A Realistic Benchmark Suite for Software Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[ZGRC14] H. Zheng, S. T. Gurumani, K. Rupnow, and D. Chen. Fast and Effective Placement and Routing Directed High-Level Synthesis for FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2014.

[ZL13] Z. Zhang and B. Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2013.

[ZLS+16] R. Zhao, G. Liu, S. Srinath, C. Batten, and Z. Zhang. Improving High-Level Synthesis with Decoupled Data Structure Optimization. *Design Automation Conf. (DAC)*, Jun 2016.

[ZMMM01] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2001.

[ZTDZ15] R. Zhao, M. Tan, S. Dai, and Z. Zhang. Area-Efficient Pipelining for FPGA-Targeted High-Level Synthesis. *Design Automation Conf. (DAC)*, 2015.